Intro to Computer Science Assignment 8

Yiping Deng

2017-11-08

1.

a. Truth Table 1

A	B	C_{in}	$A \oplus B \oplus C_{in}$
1	1	1	1
1	1	0	0
1	0	1	0
1	0	0	1
0	1	1	0
0	1	0	1
0	0	1	1
0	0	0	0

Truth Table 2

A	B	C_{in}	$(A \wedge B) \vee (C \wedge (A \oplus B))$
1	1	1	1
1	1	0	1
1	0	1	1
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	0
0	0	0	0

$$
\begin{aligned}
S &= A \oplus B \oplus C_{in} \\
&= (A \wedge B \wedge C) \vee (A \wedge \neg B \wedge \neg C) \vee \\
&\quad (\neg A \wedge B \wedge \neg C) \vee (\neg A \wedge \neg B \wedge C) \\
C_{out} &= (A \wedge B) \vee (C_{in} \wedge (A \oplus B)) \\
&= (A \vee B \vee C) \wedge (\neg A \vee \neg B \vee C) \vee \\
&\quad (\neg A \vee B \vee \neg C) \wedge (A \vee \neg B \vee \neg C)
\end{aligned}
$$

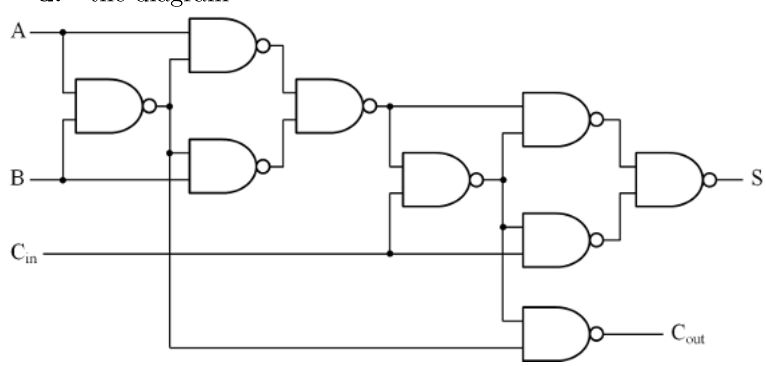
b. Just read from the truths table

$$
\begin{aligned}
S &= A \oplus B \oplus C_{in} \\
&= (\neg A \vee \neg B \vee C_{out}) \wedge (\neg A \vee B \vee \neg C) \wedge \\
&(A \vee \neg B \vee \neg C) \wedge (A \vee B \vee C) \\
C_{out} &= (A \wedge B) \vee (C_{in} \wedge (A \oplus B)) \\
&= (A \vee B) \wedge (A \vee C) \wedge (B \vee C)
\end{aligned}
$$

c. also from the truth table

$$
\begin{aligned}
S &= (A \uparrow B \uparrow C) \uparrow (A \uparrow \neg B \uparrow \neg C) \uparrow \\
&(\neg A \uparrow B \uparrow \neg C) \uparrow (\neg A \uparrow \neg B \uparrow \neg C) \\
C_{out} &= (A \uparrow B) \uparrow (A \uparrow C) \uparrow (B \uparrow C)
\end{aligned}
$$

d. the diagram



2. Code

a) implementing *bin* function below

b) implementing in the following code, use Haskell to draw truth table

c) implementing accordingly

d) same, implementing accordingly

```haskell
-- first argument is m, the length of the list
-- the second argument is n, the number in decimal

bin :: Int -> Int -> [Bool]
bin 0 _ = []
```

```haskell
bin m n = bin (m - 1) (n `div` 2) ++ [if (mod n 2) == 1 then True else False]

-- take a list of bool as a input,
-- returns a int in the end
decHelper :: [Bool] -> Int -> Int
decHelper [] x = x
decHelper (x:xs) y = decHelper xs (y * 2 + (if x then 1 else 0))

dec :: [Bool] -> Int
dec lst = decHelper lst 0

-- bit function that only works on one bit(or boolean)
-- takes the input and do the job
-- draw the entire truth table here actually
fa_c :: Bool -> Bool -> Bool -> Bool
fa_s :: Bool -> Bool -> Bool -> Bool
fa_c True True _ = True
fa_c True _ True = True
fa_c _ True True = True
fa_c _ _ _ = False
-- end of fa_c

-- simply draw a truth table here, to show what it does
fa_s False False False = False
fa_s False False True = True
fa_s False True False = True
fa_s False True True = False
fa_s True False False = True
fa_s True False True = False
fa_s True True False = False
fa_s True True True = True
-- end of truth table

-- helper function to do the carrying and sum up numbers
-- first argument is a list, representing a binary number
-- second argument is a list, a binary number
rc_helper :: [Bool] -> [Bool] -> Bool -> [Bool]
rc_helper [] xs _ = xs
rc_helper xs [] _ = xs
rc_helper (x:xs) (y:ys) c = (rc_helper xs ys (fa_c x y c)) ++ [fa_s x y c]

-- simply call the helper function
rc_add :: [Bool] -> [Bool] -> [Bool]
rc_add xs ys = rc_helper (reverse xs) (reverse ys) False

ha_c :: Bool -> Bool -> Bool
```

```haskell
ha_c x y = x && y

-- draw the truth table
ha_s :: Bool -> Bool -> Bool
ha_s True False = True
ha_s False True = True
ha_s _ _ = False

-- some helper function to sum up and do
-- the carry
calS :: [Bool] -> [Bool] -> [Bool]
calS [] xs = xs
calS xs [] = xs
calS (x:xs) (y:ys) = (ha_s x y) : (calS xs ys)

-- do carry in parallel
calC :: [Bool] -> [Bool] -> [Bool]
calC [] xs = xs
calC xs [] =xs
calC (x:xs) (y:ys) = (ha_c x y) : (calC xs ys)

-- shift the carry by one
carriesShift :: [Bool] -> [Bool]
carriesShift x = (tail x) ++ [False]

-- if it is not all zero, calculate it repeat and repeat
-- if it is, return the result:w
cla_add :: [Bool] -> [Bool] -> [Bool]
cla_add x y =
    if not $or x then y
    else if not $or y then x
    else cla_add (calS x y) (carriesShift (calC x y))
```