Computer Science - COMP3026

Assignment 1

Travis Strawbridge - University of South Australia

Email: strtk001@mymail.unisa.edu.au Student ID: 110340713

Table of Contents

Precursor - Intermediary Functions

This section presents my approach to creating the intermediary logic gate functions that the questions in this report will refer to.

As the assignment was centred around performing calculations upon binary numbers using the NOR logic gate as a base, I needed to define the *NOR* expression in a format that I could understand.

NOR expressions are equivalent to !(p||q) whereby in the assignment I apply *De Morgan's Law* to make the expression more explicit, !*p&&!q,* as I find this format more preferable.

After I roughed out the program's cycle, I began working on the addition portion of the assignment where I examined the equation for single-bit addition:

Z = (x XOR y) XOR c
C = (x AND y) OR (c AND (x XOR y)

As *NOR* expressions are functionally complete, I understood that using a combination of *NOR* I could create *NOT*, *AND* and *OR* to then create an *XOR* compound expression.

```
def NOR(x: int,y: int) -> int:
    '''
    NOR function to get the value of !x&&!y, the inverse of an OR function.
    '''
    #if x and y are false, or in this case 0:
    if x == 0 and y == 0:
        #return true, or in this case 1
        return 1
    #NOR is only true when both x and y are false so return false; 0
    else:
        return 0
```

NOT

The first compound expression I created was NOT as it was the simplest out of the group.

NOT === pNORp === !p&&!p		
p	**!p**	**!p&&!p**
1	0	0
0	1	0

By NORing p with itself we will always get the inverse value of p, due to the result of *!p&&!p*.

```
def NOT(x: int) -> int:
    '''
    Not Function that works by pushing x into NOR to get the inverse of x.

    E.g. --> x = 1;  xNORx === !x&&!x = !x = 0;
    '''
    return NOR(x,x)
```

AND

After creating NOT, AND was the next least complicated expression to create.

AND === !pNOR!q === !!p&&!!q === p&&q				
p	**q**	**!p**	**!q**	**p&&q**
1	1	0	0	1
1	0	0	1	0
0	1	1	0	0
0	0	1	1	0

NORing *!p&&!q* is the equivalent of *!!p&&!!q* which when further simplified is just *p&&q*, making the !pNOR!q only returns true when both p and q are true.

```
def AND(x: int, y: int) -> int:
    '''
    And function that works by pushing x into a NOT and Y into an NOT then pushing those values
    into a NOR the AND. We are effectively inverting the values of the NOR from !x&&!y into x&&y
    converting the NOR into an AND.

    E.g. --> x = 1, y = 1; !xNOR!y === !!x&&!!y === x&&y = 1;
    '''
    return NOR(NOT(x),NOT(y))
```

OR

The OR expression, I found, was the most difficult of the three compound expressions to create as initially I wasn't quite sure how I could possibly use NOR or my NOT and AND expressions to derive it. After incidentally creating an *equal to* expression and a lot of failed attempts, I stumbled on NORing *pNORq* with *pNORq* after trying to see how I could inverse the AND expression using NOR.

OR === (pNORq)NOR(pNORq) === !(!p&&!q)&&!(!p&&!q)						
p	**q**	**!p**	**!q**	**(!p&&!q)**	**!(!p&&!q)**	**(pNORq)NOR(pNORq)**
1	1	0	0	0	1	1
1	0	0	1	0	1	1
0	1	1	0	0	1	1
0	0	1	1	1	0	0

By inverting *(!p&&!q)* with NOT we get an non-exclusive OR expression due to De Morgan's law of *!(!p&&!q)* is equivalent to p||q.

```
def OR(x: int, y:int) -> int:
    '''
    Or function that works by getting the NOR of the NORs of x and y. This effectively inverts the
    NOR function into an OR.

    E.g:
        • OR == True:
            x = 1, y = 0; (xNORy)NOR(xNORy) === !(!x&&!y)&&!(!x&&y!);
            (!x&&!y) = 0, !(!x&&!y) = 1, !(!x&&!y)&&!(!x&&!y) = 1;
        • OR == False:
            x = 0, y = 0; (xNORy)NOR(xNORy) === !(!x&&!y)&&!(!x&&y!);
            (!x&&!y) = 1, !(!x&&!y) = 0, !(!x&&!y)&&!(!x&&!y) = 0;
    '''
    return NOR(NOR(x,y),NOR(x,y))
```

XOR

To create the XOR expression, I followed it's equivalent of *!p&&q || p&&!q* and used a compound of my NOT, AND and OR functions.

```
def XOR(x: int,y: int) -> int:
    '''
    XOR function that works by following the compound expression of !x&&y||x&&!y.
    '''
    return OR(
        AND(NOT(x),y),
        AND(x,NOT(y))
    )
```

Question 1:

"Present the truth tables for the single-bit subtraction and single-bit less-than comparison operations, and explain how did you derive them?"

Both of the truth tables for *SUB_BIT* and *LESS_THAN* are programmatically generated via a series of nested loops that apply the respective calculation process, using the range of *0..2*.

Single-Bit Subtraction Truth Table

Subtraction				
Input			**Output**	
b	**x**	**y**	**Z**	**B**
0	0	0	0	0
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	1

By following the calculation to apply single-bit subtraction to two binary numbers, we calculate the difference of *x - y* using the compound expression of *(xXORy)XORb* (which is the function *SUB_BIT*) and then check if we needed to borrow from a higher digit to get our subtraction value (using the function *BORROW_BIT*).

```
def display_subtraction_truth_table():
    '''
    Function for displaying the truth tables for the SUB_BIT and BORROW functions.
    '''
    print("SUBTRACTION\nb x y Z B\n--------")
    for in_b in range (0,2):
        for x in range(0,2):
            for y in range(0,2):
                #get the subtracted bit
                z = SUB_BIT(x,y,in_b)
                #get the borrowed bit
                out_b = BORROW_BIT(x,y,in_b)
                print(f"{in_b} {x} {y} {z} {out_b}")
    print()
```

Z Value

The *Z value* is assigned by subtracting *y* from *x* and checking if we lent the value of *x* to a lower digit. Because binary numbers only have two values (0, 1), subtracting *y* from *x* where both values are not equal to or distinct from each other, then the result will always be 0 . Conversely, if there is a difference in the values of *x* and *y* or the values are equal, the result will always be 1; *!(!x&&!y)*. Throwing *b* into the equation, we are checking if a lower digit borrowed from the current *x* value. If *x* was borrowed then *!b* = 0 and if borrowed !b = 1. So the result of *Z* = *!(!x&&!y)&&!b.*

SUB_BIT === (xXORy)XORb === !(!x&&!y)&&!b								
b	**x**	**y**	**!x**	**!y**	**(!x&&!y)**	**!(!x&&!y)**	**!b**	**!(!x&&!y)&&!b**
0	0	0	1	1	1	0	1	0
0	0	1	1	0	0	1	1	1
0	1	0	0	1	0	1	1	1
0	1	1	0	0	0	1	1	1
1	0	0	1	1	1	0	0	0
1	0	1	1	0	0	1	0	0
1	1	0	0	1	0	1	0	0
1	1	1	0	0	0	1	0	0

B Value

The *B* value is assigned by determining whether or not we need to borrow from the next higher digit during the subtraction of *x* - *y* while considering the previous borrow value *b*. If *x* = 0 and *y* = 1, then we need to borrow from the higher digit as we can't subtract a 1 from 0 without going negative. However, if both *x* and *y* are equal and *b* = 1 then we must borrow from the higher digit as in theory, x would equal 0. The *BORROW_BIT* function uses the compound expression:

B = (!x&&y) || (!(xXORy)&&b)

The first part of the expression, *(!x&&y)*, checks if a borrow is needed to subtract *y* from *x* and the second part, *(!(xXORy)&&b)*, handles the case where a previous borrow would still affect the subtraction of *x* - *y* when both *x* and *y* are both equal.

BORROW_BIT === (!x&&y) || (!(!x&&y || x&&!y),b)											
b	**x**	**y**	**!x**	**!y**	**(!x&&y)**	**!(!x&&y)**	**x&&!y**	**xXOR y**	**!(xXORy)**	**!(xXORy)&& b**	**(!x&&y)||(!(xXORy)&&b)**
0	0	0	1	1	0	1	0	0	1	0	0
0	0	1	1	0	1	0	0	1	0	0	1
0	1	0	0	1	0	1	1	1	0	0	0
0	1	1	0	0	0	1	0	0	1	0	0
1	0	0	1	1	0	1	0	0	1	1	1
1	0	1	1	0	1	0	0	1	0	0	1
1	1	0	0	1	0	1	1	1	0	0	0
1	1	1	0	0	0	1	0	0	1	1	1

Single-BIT Less-Than Truth Table

less-than			
Input			**Output**
l	**x**	**y**	**L**
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

To generate the LESS_THAN truth table we apply the *LESS_THAN* function to compare the values of *x* and *y* while considering the result of the previous comparison; *l, to get the value of L.*

The LESS_THAN function uses the compound expression of:

L = (!x&&y) || (!(xXORy)&&l)

The first part of the expression compares x and y to see if *x* = 0 and *y* = 1 or better put: if *x* < *y*. The second half checks to see if *x* and *y* are equal considering the previous comparison value of *l*. Interestingly, this is the exact same expression used in the *BORROW_BIT* function and is used for the same reason but used in different applications.

```
def display_less_than_truth_table():
    '''
    Function for displaying the truth tables for the LESS_THAN function.
    '''
    print("LESS_THAN\nl x y L\n-------")
    #construct the truth table
    for  in_l in range (0,2):
        for x in range(0,2):
            for y in range(0,2):
                #get the result of x < y
                out_l = LESS_THAN(x,y,in_l)
                #display the row
                print(f"{in_l} {x} {y} {out_l}")
    print()
```

LESS_THAN === (!x&&y) || (!(!x&&y || x&&!y),b)											
b	**x**	**y**	**!x**	**!y**	**(!x&&y)**	**!(!x&&y)**	**x&&!y**	**xXORy**	**!(xXORy)**	**!(xXORy)&&b**	**(!x&&y)||(!(xXORy)&&b)**
0	0	0	1	1	0	1	0	0	1	0	0
0	0	1	1	0	1	0	0	1	0	0	1
0	1	0	0	1	0	1	1	1	0	0	0
0	1	1	0	0	0	1	0	0	1	0	0
1	0	0	1	1	0	1	0	0	1	1	1
1	0	1	1	0	1	0	0	1	0	0	1
1	1	0	0	1	0	1	1	1	0	0	0
1	1	1	0	0	0	1	0	0	1	1	1

Question 2

"Present a NOR operation based logical expressions for the single-bit addition, single-bit subtraction and single-bit less-than comparison, and explain how you derived them"

Single Bit Addition

As mentioned in the *Precursor section* of this report, I created NOR based logical expressions for NOT, AND, OR and XOR. This was so I could follow the equation for *single-bit addition* of:

Z = (xXORy) XOR c

By creating the expressional functions via compounds of NOR and NOR based expressions, I was able to implement the *SUB_ADD* function to achieve single-bit addition based on the equation for addition that was provided in the spec.

```
def ADD_BIT(x: int,y: int,c: int) -> int:
    '''
    Single bit addition function that sums the x and y values considering the carry over from the previous
    LSD bit; c
    '''
    return XOR(XOR(x,y),c)
```

Single-Bit Subtraction

As explained in the *Z Value section of Question 1 (Single-Bit Subtraction Truth Table)*, the SUB_BIT function works by subtracting *y* from *x* and checking if we lent the value of *x* to a lower digit. To do this we need an expression to subtract the value of *x* by *y* where *x* and *y* don't equal 0 (as we don't care about 0 - 0). We also need an expression to check if the value of *x* has been borrowed by a lower digit; where, essentially, *x* would then equal 0.

Z = (xXORy)XORb === (x || y)&& !b

The first expression checks if x and or y are true and that we haven't lent the current *x* value to a lower digit. Also just like how the *BORROW_BIT* and *LESS_THAN* functions are the same expression used in different applications, the *single-bit addition* works the same way as the single-bit subtraction.

```
def SUB_BIT(x: int, y: int,b: int) -> int:
    '''
    Single bit subtraction function that calculates the difference of x - y,
    while considering the borrow value, b ,from the previous LSD.
    '''
    return XOR(XOR(x,y),b)
```

Single Bit Less-Than Comparison

As explained in the *Single-Bit Less-Than Truth Table section of Question 1,* we needed an expression that would check if $x < y$ while considering the previous comparison result; *l.*
To achieve this we used the compound expression:

L = (!x&&y) || (!(xXORy)&&l)

The first part of the expression, *(!x&&y)*, is effectively: $x < y$ where $x = 0$ and $y = 1$. The second part, *(!(xXORy)&&l),* is checking if *x* and *y* are equal and if they are, then the result depends on whether or not a lower digit had already been considered less-than. If *l* is true, then that result is carried forward to the next highest digit.

```
def LESS_THAN(x: int, y: int, l: int) -> int:
  '''
  Single bit comparason function for comparing if x < y, considering the result of the lower digit from
  a previous check; l.
  '''
  return OR(
    AND(
      NOT(x),
      y
    ),
    AND(
      NOT(XOR(x,y)),
      l
    )
  )
```

Question 3

“Describe and explain how you perform multi-digit binary number addition, subtraction, and comparison, using the single-bit operation functions.”

Multi-Digit Addition

To perform multi-digit addition using the binary lists, we first need to store the max length between *binary_list_x* and *binary_list_y* so that we have a boundary for later iterating over both lists.
We then create copies of the lists so we don't modify them and can use them to display the values later.
Next we reverse the item order of both lists so that we can iterate from the least significant digit (*LSD*) to the most significant digit (*MSD*) or from right to left.
We then need to create a list to hold the summed values of our calculation and also create a variable to store the carryover bit value.
After that we iterate over the max length we defined earlier in the first step so that we can get the bit value of each list at the current index (0 if the current index is out of bounds of one of the lists) then append the result of *ADD_BIT*(*item_x*, *Item_y*, *carryover_value*) to the sum list and set the next value for the carryover bit.
Once the iteration cycle is complete, we then check if the carryover value equals 1 and if it does then we append that to the sum list as this is the final bit of the sum of *binary_list_x* + *binary_list_y.*
We now need to reverse the sum list so that the MSD is at index 0 and the LSD is at index *len(summed_list)-1*
Then we finish by making a call to display the resulting equation.

```
def addition(binary_x: list[int], binary_y: list[int]):
    '''
    Function that performs the addition operation of binary_x + binary_y
    '''
    #get the max_len between both lists
    max_len = max(len(binary_x),len(binary_y))
    #create copies of the binary lists so we dont modify the originals
    list_x = list(binary_x)
    list_y = list(binary_y)
    #reverse the binary lists so that we are dealing with the least significant digit in both lists
    list_x.reverse()
    list_y.reverse()
    #declare a binary list to hold the sum of the x and y lists
    binary_sum = list()
    #declare a var to hold the bits to be carryover over
    carryover = 0
    #interate over the max_len of the two binary lists in reverse so we start from the
    #least significant digit to the highest
    for i in range(0,max_len):
        #initialise x and y as 0 as a default incase either binary list is shorter in length
        x = 0
        y = 0
        #check the list lengths to see if i is within bounds and set x or y to the item at
        #the current index
        if i < len(list_x):
            x = list_x[i]
        if i < len(list_y):
            y = list_y[i]
        #append the sum bit to the binary sum list
        binary_sum.append(ADD_BIT(x,y,carryover))
        #calculate the carryover bit
        carryover = CARRY_BIT(x,y,carryover)
```

```
#check if the carryover bit for the final addition is 1
if carryover == 1:
    binary_sum.append(carryover)

#reverse the binary sum list as we added the values in backwards from right to left
binary_sum.reverse()
#display the equation
display_equation(binary_x,binary_y,binary_sum,"+")
```

Multi-Digit Subtraction

To perform multi-digit subtraction:
We get the max length of both binary lists so we later have a boundary to iterate until.
We then make a call to the *is_less_than* function to return a bool if *binary_list_x < binary_list_y* as we can later determine if the resulting number will be negative if *y > x*.
Create copies of the lists so we don't modify the original lists.
We then check if the resulting number from this calculation will be negative and if true we swap the lists around so that *list_x = list_y* and *list_y = list_y.* Switching the lists makes it so we can subtract the smaller number from the bigger number and throw a negative sign in front of the resulting number when we display the equation.
After the negative result check, we reverse the lists so that the LSD is index 0 and the MSD is index *list_length*.
Next we create a list to hold the resulting values from the calculation and create a var to hold the burrowed bit.
Then we iterate over each list *max_length* times, grabbing the values from each list at the current index (0 if we have gone out of bounds for a given list) then append the result of *SUB_BIT*(*list_x, list_y, borrowed_bit*) to the result list and set the next value of the borrowed bit.
After the iteration cycle has finished, we then check if we still have a borrowed bit (*borrowed_bit == 1)* and append it to the result list as that is the final bit to be added to the result list.
We then reverse the result list so that the MSD is at index 0 and the LSD is the last item in the list.
Then we need to trim any trailing 0's from the result if there is any and then finally;
we can display the resulting equation.

```python
def subtraction(binary_x: list[int], binary_y: list[int]):
    '''
    Function that performs the subtraction operation of binary_x - binary_y
    '''
    #get the max_len between both lists
    max_len = max(len(binary_x),len(binary_y))
    #check if binary_x is less than binary_y which would result in a negative result
    is_negative = is_less_than(binary_x,binary_y)
    #create copies of the binary lists so we dont modify the originals
    list_x = list(binary_x)
    list_y = list(binary_y)
    #if the number will be negative, flip the lists so that x == y y == x
    if is_negative:
        list_x,list_y = list_y,list_x
    #reverse the lists
    list_x.reverse()
    list_y.reverse()
    #declare a binary list to hold the result of the subtraction of x - y
    binary_result = list()
    #declare a var to hold the bits to be borrowed
    borrowed = 0
    #iterate over the binary lists from LSD to MSD (right to left)
    for i in range(0,max_len):
        #initialise x and y as 0 as a default incase either binary list is shorter in length
        x = 0
        y = 0
        #check the list lengths to see if i is within bounds and set x or y to the item at
        #the current index
        if i < len(list_x):
            x = list_x[i]
        if i < len(list_y):
            y = list_y[i]
        #append the resulting bit to the binary result list
        binary_result.append(SUB_BIT(x,y,borrowed))
        #calculate the carryover bit
        borrowed = BORROW_BIT(x,y,borrowed)

    #check if the borrowed bit for the final subtraction is 1
    if borrowed == 1:
        binary_result.append(borrowed)

    #reverse the binary result list
    binary_result.reverse()
    #remove the trailing 0's if there are any
    for i in range(0,len(binary_result)):
        #if the item of index i is 1 then slice the list from here
        if binary_result[i] == 1:
            binary_result = binary_result[i:]
            break

    #display the equasion
    display_equation(binary_x,binary_y,binary_result,"-",is_negative)
```

Multi-Digit Comparison

To perform multi-digit comparison of the two binary lists:
We get the max length between both binary lists just like the addition and subtraction processes.
We then create copies of each list but we add *n = max_length - list_length* trailing 0's to each list so that if one number is larger than the other we can still compare the smaller number with the larger number but without bounds checks.
Next we reverse both lists.
We declare a var, *l*, to hold the result from the future *LESS_THAN*() results
We then iterate over the lists *max_length* times, getting the items of both lists at the current index and setting the value of *l* to the result of *LESS_THAN(item_x, item_y, previous_less_than).*
Finally after completing the loop, we then return *l == 1* so that if the final digit of *list_x* is less-than the final digit of *list_y* we return true, and obviously, false if the last digit of *list_y* is greater-than.

```
def is_less_than(binary_x: list[int],binary_y: list[int]) -> bool:
    '''
     function for performing multi bit comparason to see if binary_x is < binary_y and returns a bool
    '''
    #get the max_len between both lists
    max_len = max(len(binary_x),len(binary_y))
    #initalise copies of the lists so we dont work with the references
    x_list = list([0]*(max_len - len(binary_x)) + binary_x)
    y_list = list([0]*(max_len - len(binary_y)) + binary_y)

    #reverse both lists to compare from LSD to MSD
    x_list.reverse()
    y_list.reverse()

    l = 0
    #iterate over each item in both lists
    for i in range(0,max_len):
        #get the items
        x = x_list[i]
        y = y_list[i]
        #compare x and y to determine if x < y
        l = LESS_THAN(x,y,l)
    #return true if x < 1 or false if not
    return l == 1
```

Question 4

“Present output of your program that answers the following questions…”

1101011000 + 1011010100 = ?:

```
Choose operation [+, -, q]: +
X: 1101011000
Y: 1011010100

   1101011000
+  1011010100
-------------
  11000101100
```

1101110000 – 110101010 = ?:

```
Choose operation [+, -, q]: -
X: 1101110000
Y: 110101010

  1101110000
-  110101010
------------
   111000110
```

111000110 – 1101110000 = ?:

```
Choose operation [+, -, q]: -
X: 111000110
Y: 1101110000

   111000110
- 1101110000
------------
  -110101010
```

1111100000111110000011111000001111100000111110000011111000001111100000 + 11111000001111100000111110000011111000001111100000111110000011111 = ?:

```
Choose operation [+, -, q]: +
X: 1111100000111110000011111000001111100000111110000011111000001111100000
Y: 11111000001111100000111110000011111000001111100000111110000011111

  1111100000111110000011111000001111100000111110000011111000001111100000
+      11111000001111100000111110000011111000001111100000111110000011111
------------------------------------------------------------------------
  1111111111111111111111111111111111111111111111111111111111111111111111
```

1111100000111110000011111000001111100000111110000011111000001111100000 − 11111000001111100000111110000011111000001111100000111110000011111 = ?

```
Choose operation [+, -, q]: -
X: 1111100000111110000011111000001111100000111110000011111000001111100000
Y: 11111000001111100000111110000011111000001111100000111110000011111

  1111100000111110000011111000001111100000111110000011111000001111100000
-      11111000001111100000111110000011111000001111100000111110000011111
------------------------------------------------------------------------
  1111000001111100000111110000011111000001111100000111110000011111000001
```