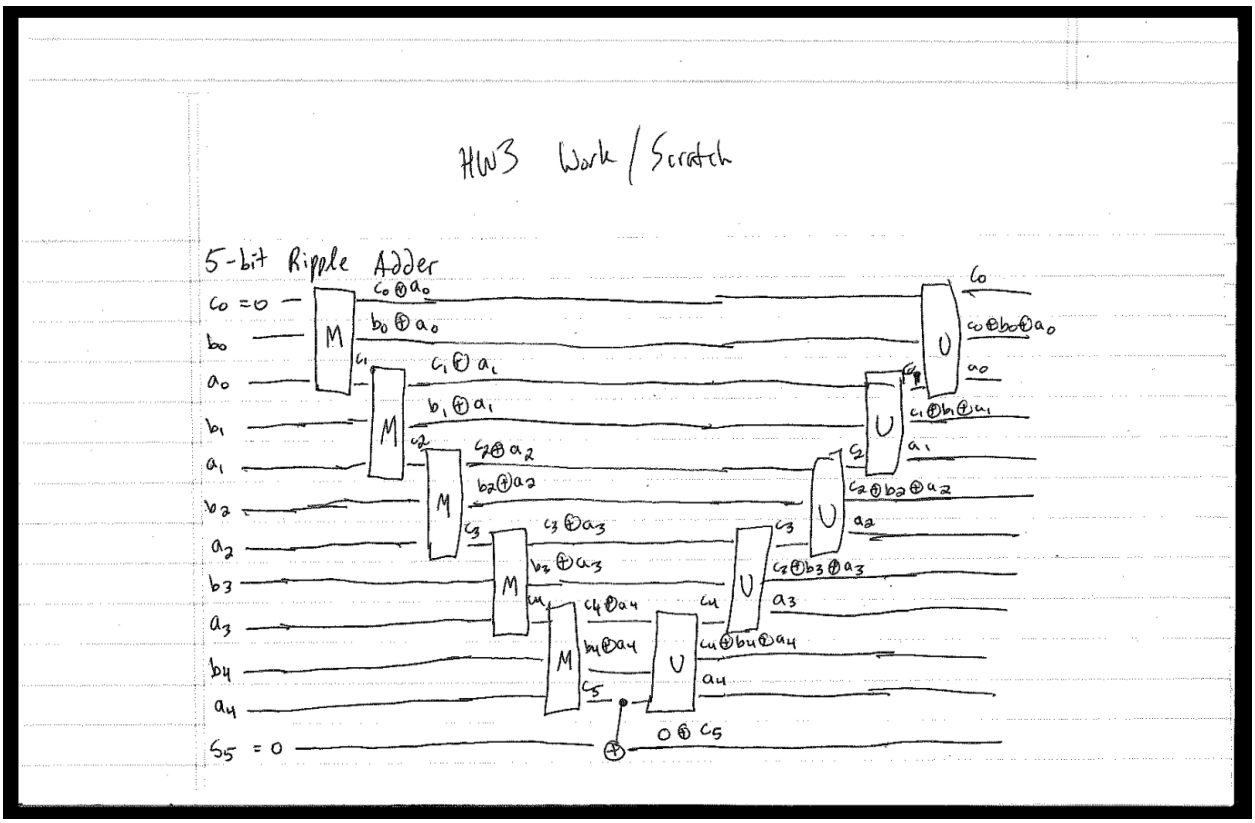
Quantum Programming HW3

B490 : Spring 2020

Joshua Larkin

1 Exercise 1

1. Here is my diagram: M means 3-bit majority and U means 3-bit unmajority-and-add



2. Included in a file titled `adder_n.txt`

3. Included in a file titled `adder.py`

2 Exercise 2

1. Convert the truth tables for the following functions/circuits into cyclic notation: For this, we will give each element of the given context set (n-bit) a unique letter, and the cycle notation for transpositions of elements will use the letters.

   - x (in a 1-bit context)

     x on 1 bit is just $(a\ b)$

_	a	$x\ a$	_
(a)	0	1	(b)
(b)	1	0	(a)

   - x (in a 4-bit context)

     The transposition here is $(a\ i)(b\ j)(c\ k)(d\ l)(e\ m)(f\ n)(g\ o)(h\ p)$

_	a	$x\ a$	_
(a)	0000	1000	(i)
(b)	0001	1001	(j)
(c)	0010	1010	(k)
(d)	0011	1011	(l)
(e)	0100	1100	(m)
(f)	0101	1101	(n)
(g)	0110	1110	(o)
(h)	0111	1111	(p)

_	a	$x\ a$	_
(i)	1000	0000	(a)
(j)	1001	0001	(b)
(k)	1010	0010	(c)
(l)	1011	0011	(d)
(m)	1100	0100	(e)
(n)	1101	0101	(f)
(o)	1110	0110	(g)
(p)	1111	0111	(h)

- cx (in a 2-bit context)

  The transposition is: $(c\ d)$

_	a	$cx\ a$	_
(a)	00	00	(a)
(b)	01	01	(b)
(c)	10	11	(d)
(d)	11	10	(c)

- cx (in a 4-bit context)

  The transposition is: $(i\ m)(j\ n)(k\ o)(l\ p)$

_	a	$cx\ a$	_
(a)	0000	0000	(a)
(b)	0001	0001	(b)
(c)	0010	0010	(c)
(d)	0011	0011	(d)
(e)	0100	0100	(e)
(f)	0101	0101	(f)
(g)	0110	0110	(g)
(h)	0111	0111	(h)

_	a	$cx\ a$	_
(i)	1000	1100	(m)
(j)	1001	1101	(n)
(k)	1010	1110	(o)
(l)	1011	1111	(p)
(m)	1100	1000	(i)
(n)	1101	1001	(j)
(o)	1110	1010	(k)
(p)	1111	1011	(l)

- *ccx* (in a 3-bit context)

  The transposition is: $(g\ h)$

_	a	$cx\ a$	_
(a)	000	000	(a)
(b)	001	001	(b)
(c)	010	010	(c)
(d)	011	011	(d)
(e)	100	100	(e)
(f)	101	101	(f)
(g)	110	110	(h)
(h)	111	111	(g)

- *ccx* (in a 4-bit context)

  The transposition is $(m\ o)(n\ p)$

_	a	$cx\ a$	_
(a)	0000	0000	(a)
(b)	0001	0001	(b)
(c)	0010	0010	(c)
(d)	0011	0011	(d)
(e)	0100	0100	(e)
(f)	0101	0101	(f)
(g)	0110	0110	(g)
(h)	0111	0111	(h)

_	a	$cx\ a$	_
(i)	1000	1000	(i)
(j)	1001	1001	(j)
(k)	1010	1010	(k)
(l)	1011	1011	(l)
(m)	1100	1110	(o)
(n)	1101	1111	(p)
(o)	1110	1100	(m)
(p)	1111	1101	(n)

2. Prove that the function CCCNOT cannot be implemented with a 4-bit circuit using the gate basis $\{\texttt{x}, \texttt{cx}, \texttt{ccx}\}$

   Proof. We know from our above work that each of the gates, in a 4-bit context, are transpositions of even parity. By the given theorem, we know that any composition of the gates in the gate basis will have an even length. Consider the CCCNOT function: it will swap 1110 and 1111, and leave all other inputs fixed. This means that CCCNOT is essentially a transposition of odd parity. Hence we cannot implement CCCNOT using the gate basis $\{\texttt{x}, \texttt{cx}, \texttt{ccx}\}$ □

3. Prove that a 5-bit majority function cannot be implemented with a 5-bit circuit using the gate basis $\{\texttt{x}, \texttt{cx}, \texttt{ccx}\}$

   Proof. We claim that the basis gates are still transpositions of even parity in a 5-bit context. In the case for `x`, we will swap all 32 rows, which is 16 transpositions. In the case for `cx`, we will swap 16 rows, which is 8 transpositions. In the case for `ccx` we will swap 8 rows, which is 4 transpositions. We claim that the majority function has 10 row swaps, and is thus a 5 transposition function. The majority function will have one of the bits represent the majority, so there will be some column of the output that has a 1 when there is a majority. We may assume without loss of generality that this is the first column of the output. Of the first 16 rows of possible inputs, there are 5 that will need to be swapped: 00111, 01011, 01101, 01110, 01111. And becaues of the symmetry of 5 bits, there will be 5 more in the last 16 rows. So we know majority is a odd parity transposition function, and our gate basis is only made up of even parity transposition functions, so there is no way to implement majority with the gate basis $\{\texttt{x}, \texttt{cx}, \texttt{ccx}\}$ □

3 Exercise 3

Define `toffoli`(n) in Π.

```
tof :: (n : Nat) -> B^n <-> B^n
tof 0     = id
tof 1     = not
tof 2     = distrib; ((id x not) + id); factor
tof n + 1 = distrib; (tof n); factor
```

4 Exercise 4

Define `toffoli`(n) in Thesus. First recall these definition from Theseus:

```
not :: Bool <-> Bool
| True  <-> False
| False <-> True

if :: th:(a <-> a) -> el:(a <-> a) -> Bool x a <-> Bool x a
| True,  a <-> True, th a
| False, a <-> False, el a

cx :: Bool x Bool <-> Bool x Bool
| x <-> if ~th:not ~el:id x
```

Now we can define Toffoli:

```
tof :: (n : Nat) -> B^n <-> B^n
| 0     <-> id
| 1     <-> not
| 2     <-> cx
| n + 1 <-> if ~th:(tof n) ~el:id
```