Nicole Nechita - rone8293
Simon Nilsson - sini3794

# Documentation of Modular Exponentiation

## 1.1 Set_Bits

Set bits takes a circuit, a register or list of qubits and a binary string, and in turn copies a said binary string into the register by applying X gates. It is expected that the length of the register or qubits to be equal to the length of the binary string.

Because one may insert a binary string where the left bit is the most significant, it was necessary to use string slicing to properly copy the binary string, or risk inserting them backwards, because in the circuit, len(A)-1 is the most significant bit.

Using this function again with the same parameters sets the register, or qubits, back to their previous state.

During tests, this function has run without issue up until register a is approximately 12 qubits. The function slows down by then and will eventually cast error for exceeding qubit limits on the circuit.

```
7   #the set_bits function 1.1
8   def set_bits(circuit, a, x):
9       x = x[::-1]
10      for i in reversed(range(len(a))):
11          if x[i] == "1":
12              circuit.x(a[i])
```

Figure 1. Set_bits snippet

## 1.2 Copy

This function takes a circuit, and either two registers or two lists of qubits. It will proceed to apply a controlled X gate with control bits in register a and target bit in register b. This will effectively copy values from register a to b, if we assume both are the same length. This function can be called again to return register b to its default state.

During testing, this function had no issue with different lengths of bits in the registers, but eventually would slow down due to qubit limitations on circuit.

```
13  #the copy function 1.2
14  def copy(circuit, A, B):
15      amount_registers = len(A)
16      for i in range(amount_registers):
17          circuit.cx(A[i],B[i])
```

Figure 2. Copy function

## 1.3 Full Adder

This function takes in a circuit, qubits a and b, a qubit to store sum, a qubit which stores the carry in value, a qubit which carries the carry out value and an auxiliary qubit used for calculations. This function is primarily used to perform addition in larger numbers one bit after another using a series of controlled X and Toffoli X gates, and carries carry in and carry out values when appropriate.

The auxiliary qubit is reset to its default state after performing the same operations on it again at the end of the function. Likewise, performing the function again on the same parameters sets sum and carry out qubits back to their default state, even if it doesn't serve a use.

Because of the nature of the function,  there aren't many tests, and adding more qubits is not possible, no slowdowns possible due to this.

## 1.4 Addition

This function takes a circuit, either two registers or two lists of qubits, a register or list of qubits where sum is stored and a register or list of auxiliary qubits used for calculations. For the sake of the operation, 6 auxiliary qubits are needed to perform the necessary calculations. Performing this function again with the same parameters sets the register or qubits used to store the sum back to its default state. After addition is complete, the auxiliary qubits are set to their default state

This function employs the use of the full adder function by bitwise performing it and taking the value of carry out qubit to the next bit when appropriate.

Because the register or list of qubits used for the result remains the same size, operations such as 1000 + 1000 will lead to 0000 instead of 10000. This can lead to strange behaviour if tried with modulus.

Under tests, this function has no issue with any variation in length, but just like other functions, increasing the number will make the operation slower and slower before reaching the limit for a circuit.

```
48  def addition(circuit, a, b, r, aux):
49      #do the calculation
50      for i in range(len(a)):
51          full_adder(circuit, a[i], b[i], r[i], aux[i+1], aux[i+2], aux[0])
52      # reset aux
53      for i in reversed(range(len(a))):
54          circuit.ccx(a[i],b[i], aux[i+2])
55          circuit.cx(a[i],b[i])
56          circuit.ccx(b[i],aux[i+1],aux[i+2])
57          circuit.cx(a[i],b[i])
```

Figure 3. Addition Function

## 1.5 Subtraction

This function takes a circuit, either two registers or two lists of qubits, a register or list of qubits to store subtraction results, and a register or list of auxiliary qubits used for calculations mid function. This function works by flipping all values in the second register by applying X gates, and then performing an addition of the first register with this flipped second register. The second register is unflipped after the operations.

The auxiliary qubits are used for the addition function, so 6 are required. This also means that the auxiliary qubits are in their default state by the end of the function. Performing this function again will set the result register to its default state.

Just like addition, trying to perform subtraction with differing number lengths poses no issue, except when the numbers become large and the operation slows down, or can't perform due to limitations.

```
67  def subtraction(circuit, a, b, r, aux):
68      # flip b and set first carry_in to 1
69      circuit.x(b)
70      circuit.x(aux[1])
71      # do addition
72      addition(circuit, a, b, r, aux)
73      # rest aux
74      circuit.x(b)
75      circuit.x(aux[1])
```

Figure 4. Subtraction function

## 1.6 Greater than or equal

This function works with the use of subtraction or rather the carry out bit from subtraction. In this qubit the result, 1 or 0 will be placed in the result qubit. So this circuit works in the same way that subtraction does, it just removes the gates that calculates the actual result from the subtraction and moves the final carry over qubit into the result qubit before resetting the auxiliary qubits to their default state. Performing this function again will set the result register or qubit back to its default state

The carry over circuit is made with a loop that just copies the part of the circuit that does the carry out logic in the 1.3 full adder.

```
circuit.x(b)
circuit.x(aux[0])
# claculate the carry_out
for i in range(len(a)):
    circuit.ccx(a[i],b[i], aux[i+1])
    circuit.cx(a[i],b[i])
    circuit.ccx(b[i],aux[i],aux[i+1])
    circuit.cx(a[i],b[i])
    circuit.barrier()
```

Figure 5. Greater than or equal snippet

During tests this one is much alike to other functions, where differing numbers work without issue, but large sizes lead to slowdowns.

Because this function is based on the subtraction and in turn the addition function it does require an increasing amount of aux qubits, each carry out bit in the calculation gets an qubit. It is a slight improvement, compared to addition, as it does not require the one qubit that is required to calculate the result in addition, and subtraction.

## 1.7 add mod

This function makes use of custom controlled gates for subtraction and copy. Otherwise nothing really different happens in this function. The addition is performed and the result is placed temporarily in specified auxiliary qubits then it is compared to the value of n and if it's greater than n, then n will be subtracted from it and placed in the result register, otherwise copy the result of A+B into the result register (r) and then reset the aux.

It is worth noting that when A and B happen to be added in such a way that ex: 1000 + 1000 the result will be 0000, which will give strange reactions with modulo, say if N was 1100. Normally in decimals, this would mean 8+8 = 16 - 12 = 4 = 0100(binary). But because of the way addition works, that is not possible.

Using the to_gate() function creates simple gates which act as copy and subtraction but with a control qubit, compared to more proper custom gates that are defined using classes. We argue these save on energy and are essentially functions but with a control qubit and do not count as breaking the rule stating we shouldn't gates other than X,controlled X, toffoli X and multi controlled X.

```
def add_mod(circuit, n, a, b, r, aux):
    #make controled gates
    qcs_a = QuantumRegister(len(a), "a")
    qcs_b = QuantumRegister(len(a), "b")
    qcs_r = QuantumRegister(len(a), "r")
    qcs_aux = QuantumRegister(len(a)+2,"aux")
    qcs = QuantumCircuit(qcs_a,qcs_b,qcs_r,qcs_aux)
    subtraction(qcs, qcs_a,qcs_b,qcs_r,qcs_aux)
    sub_gate = qcs.to_gate(None, "mysub").control(1)
    qcc_a = QuantumRegister(len(a), "a")
    qcc_b = QuantumRegister(len(a), "b")
    qc = QuantumCircuit(qcc_a,qcc_b)
    copy(qc, qcc_a, qcc_b)
    copy_gate = qc.to_gate(None, "mycopy").control(1)
```

Figure 6. Custom gate used in 1.7 and 1.8

Regarding the modulo calculation we have it define a part of the aux register with the same size as a and b to be used to store the qubits needed to set the n value. After setting these qubits to the right values it compares it with the result from the addition which also takes up a similarly sized partition of the aux register.  We then apply the custom control gates we previously declared but place a not gate on the control qubit in between them so that whenever one of the control gates is executed the other does not.

```
#modulo
set_bits(circuit, n_qbits, n)
greater_than_or_equal(circuit, add_r, n_qbits, aux[0], rest_aux)
# do subtraction if add_r is grater then n
circuit.append(sub_gate, get_qbits([aux[0]], [add_r, n_qbits, r, rest_aux]))
# copy add_r into r if add_r is lees than n
circuit.x(aux[0])
circuit.append(copy_gate, get_qbits([aux[0]], [add_r, r]))
```

Figure 7. The modulo circuit used in 1.7

Because this function partitions aux into 2 additional value spaces this function will take up two times the length of the value a/b plus the aux required for that length from the addition function and plus one additional bit to use as the control bit for the custom gates. This results in this circuit needing aux qubits with the minimum size of 3 times the length of a/b, plus 3.

Testing results in as expected higher simulation times then the ones from previous stages in the project. With 5 bits or higher the resulting circuit exceeds the roof of what the aer simulator can handle making them impossible to test. Testing with numbers of 4 bits long is slow, but doable and gives expected results.

## 1.8 times two mod

This function is in principle similar to 1.7. The only difference and the reason why the 1.7 function is not called in this function is because the lack of space needed for a different register for a different number B allows space to be reused in the aux register between the operations. Happening. Because this circuit is len(A) qubits smaller than 1.7 this function is more space efficient than it otherwise would be and thus faster than 1.7, otherwise they are the similar in structure.

It is worth noting that when A happens to be doubled in such a way that ex: 1000 + 1000 the result will be 0000, which will give strange reactions with modulo, say if N was 1100. Normally in decimals, this would mean 8+8 = 16 - 12 = 4 = 0100(binary). But because of the way addition works, that is not possible.

During testing, having larger numbers slows down the computation significantly or casts exceptions, but it works as expected when it does work.

```
# claculate the result of a * 2
copy(circuit, a, extra_value)
addition(circuit, a, extra_value, add_r, rest_aux)
copy(circuit, a, extra_value)
```

Figure 8. The addition function in 1.8 where the second input value is freed up to be used later to save on space.

```
# modulo clauculation
set_bits(circuit, extra_value, n)
```

Figure 9. Reassignment of qubits so they can be used in the modulo circuit.