# Computing width and depth of a quantum program in the Tracer

This document defines the terms and describes our approach to computing depth and width of a quantum program (circuit) in the tracer component of the quantum execution environment (Q# runtime). It also describes the limitations and a possible way to overcome them.

## Terms

### Quantum Program

Quantum program is a program written in a Q#. Q# is a high-level language and typically doesn’t restrict decompositions, specific order of execution, etc. It operates in terms of user qubits. Quantum program can be translated to a quantum circuit.

### Quantum Circuit

Quantum circuit is a specific way of executing quantum program. It specifies order of gates, which gates can be executed in parallela and which underlying qubits are used for specific user qubits.

### Tracer

Tracer is a component of a quantum execution environment that allows “execution” of a quantum program without a quantum device and without complete simulation of a quantum device on classical hardware.

The word execution here should not be taken literally because measurements will not have actual results that would be obtained on a quantum system, instead they are mocked in certain way to allow quantum program to proceed. With this approach tracer can execute quantum programs much faster than the complete honest simulator would (full-state simulator) and collect statistical data about it. This is useful for resource estimation.

### Qubit Manager

Qubit manager is a component of a quantum execution environment that maps program demand for qubits (user qubits) to available qubits in the system (underlying qubits).

Quantum program may allocate and deallocate qubits just like classical program allocates and deallocates classical memory. It is up to qubit manager to decide how to serve such allocation requests and whether to reuse previously released underlying qubits or use fresh ones.

### Circuit Width

Width of a quantum circuit or a quantum program, roughly defined, is the number of qubits needed to execute it.

There’re several uncertainties and details in this definition that need to be clarified. First, we cannot statically analyze all branches of a program and make conclusions about its width and depth in all possible cases unless constraints or assumptions are in place. So, we measure width of one specific execution of a quantum program – width of a specific circuit. Second, we count qubits that the circuit uses while executing on a Tracer. Execution on a quantum device may require other layers between the actual quantum device and the program such as a fault-tolerant encoding, which could increase number of actual qubits significantly. This increase is out of scope of Tracer.

### Circuit Depth

Depth of a quantum circuit or a quantum program, roughly defined, is the time it takes to execute it measured in appropriate units.

Only quantum part is measured. We don’t take into account classical computation even if it takes considerable time. Often the time is quantized, and the unit of measurement is one time step. Often many quantum gates can be executed in parallel. In this case we don’t add up all their times to get the depth. For example, if five gates each taking two ticks are executed in parallel, the depth of such circuit is still two time steps. In some cases we want to measure specific depth, such as T-Depth. In this case all gates except T gates as considered to have execution time zero.

## Objective

Because quantum program can be represented by different circuits (executed in different ways), such circuits may have different widths and depths. Historically one pair of numbers were collected – minimum width and minimum depth. These numbers were not compatible, i.e. it wasn’t possible to achieve both with the same circuit. Such results were not sufficient, so the objective changed.

The objective is to collect two pairs of numbers. First pair –low width and corresponding depth. To compute this pair we try to execute a program in such a way that it would take least amount of qubits and then measure the number of time steps required to execute it.

Second pair – low depth and corresponding width. To compute this pair we try to execute a program in such a way that it would take least amount of time steps and then count the number of qubits needed to execute it in such way.

As mentioned before, we cannot statically analyze a program and deduce its behavior in all possible cases. We also do not automatically write a different but equivalent program which would have the optimal depth or width. We can only measure one execution of a program. That said we do make equivalent transformations to the specific execution of a program to obtain desirable statistics. For example, we may decide to execute multiple gates in parallel if it doesn’t change the result. Such parallel execution will result in lower depth. Specific approach will be described later.

## Role of Qubit Manager

Qubit manager plays crucial role in width and depth calculation via qubit reuse. Consider the following example (a and b are different qubits):

Allocate(q1)  
Gate1(a, q1)  
Deallocate(q1)  
Allocate(q2)  
Gate2(b, q2)  
Deallocate(q2)

Gates Gate1 and Gate2 operate on different qubits so they can be executed simultaneously. Assuming it takes one time step to execute Gate1 and Gate2, executing them simultaneously also takes one time step. That is if we also assume qubits q1 and q2 are different. However, this program deallocates qubit q1 before allocating qubit q2. Qubit manager may decide to reuse the same underlying qubit for both names. This will result in the following transformation:

Gate1(a, c)  
Cleanup(c)  
Gate2(b, c)

These two gates operate on the same qubit c and therefore Gate2 can only be executed after Gate1. They no longer can be executed at the same time. Executing Gate1 and Gate2 one after another takes two time steps. So we see that reusing qubits increases the depth of the quantum circuit.

Since different qubit manager strategies affect reuse of qubits, we employ two different modes of qubit manager to obtain two different extremes – low depth and low width. The strategy choice is controlled via EncourageReuse parameter.

## “Gravity” algorithm for obtaining minimum depth.

We associate availability time with each qubit . In the beginning, each is zero for all qubits. For every n-qubit gate executed by the quantum program we update availability time associated with its argument qubits as follows:

Where is the depth of the gate. In the end we take the maximum among availability times associated with all qubits in the system and declare it the minimum depth.

This is a greedy linear algorithm in the number of gates (assuming gate arity is limited by a global constant, which is equal to 3 or 4 in practice). It can be understood as executing gates as early as possible. It can be visualized as applying “gravity” to the circuit and shaking gates as close to the beginning as possible. The algorithm reorders gates in the quantum program; it schedules each gate according to its time.

## Low Width

Algorithms that provide minimal width with gate reordering to achieve it are computationally intensive and are out of scope of current implementation. Therefore, a greedy approach is employed. Greedy algorithm achieves minimal depth assuming gates are **not** reordered.

First, we configure qubit manager to reuse of qubits (EncourageReuse = true). This gives us a reasonably low number of qubits used, hence the low width of a circuit. We will not reorder gates to obtain better width. As a result, obtained width is a reasonable upper bound on the minimum width.

Note that the ordering in the program is sequential, therefore the depth without reordering is always equal to the number of gates. Once the width is obtained, we employ greedy “gravity” algorithm to reorder gates to execute them as early as possible. Then we compute depth of resulting circuit.

This will result in a pair of numbers – low width, which is a reasonable upper bound on minimal width and a low depth, compatible with this depth. We call them **low width and compatible depth**.

## Minimum Depth

To obtain minimum depth we configure qubit manager not to reuse qubits (EncourageReuse = false. When qubits are not reused, no additional dependencies are introduced (beyond ones that are present in the quantum program) and more gates can be executed in parallel.

Then we employ greedy “gravity” algorithm to reorder gates and execute them as early as possible. The resulting circuit gives us the minimum depth and corresponding compatible width. Depth is guaranteed to be minimal. Although simple to compute, the width may not be the lowest **given the minimum depth** of the circuit. In other words, we may reorder some gates, reuse some qubits, and still get the minimum depth with lower width. Most of these approaches are computationally intensive and is out of scope of current implementation.

One approach that can be employed is the greedy coloring of an interval graph described here: <https://en.wikipedia.org/wiki/Greedy_coloring>. This approach will not perform any reordering to minimize width given the depth. Instead, the ordering will be the same as in minimal depth calculation, but it will compute minimum width assuming this ordering (and keep the depth unchanged). This is an offline algorithm, meaning it can only obtain results at the end of a quantum program. It may be desirable to obtain results for every scope right after exiting of each scope.

The approach we use is a set of heuristics and an on-line algorithm that is described later. It results in a pair of numbers – minimal depth and a reasonable width, compatible with the depth. Calculated width is an upper bound on minimum width, given the minimum depth constraint. We call them **minimum depth and compatible width**.

## Allocation/Deallocation optimization

One further optimization that is employed is allocation/deallocation time optimization. Such optimization will ignore specific times of user allocation and deallocation of qubits, instead such times will be derived from first and last use of the qubits. For example, at some point in time t1 user may be done with qubit q1. Then several gates are executed on other qubits and then qubit q1 is deallocated at time t2 > t1. In this case user qubit q1 can be deallocated at time t1 rather than t2 and underlying qubit can be reused earlier.

## Parameters for width and depth calculations

Besides quantum program itself, the following parameters influence measured width and depth.

### Single gate times

It may take different time to execute different gates. While the simplest case is when every gate takes one time step, such approach isn’t always sufficient to evaluate a circuit. Currently, a floating-point number representing gate’s execution time should be provided for each type of gate.

Providing a floating point gate time presents a challenge for qubit reuse based on availability time and reordering. As computations with floating point numbers aren’t precise, subsequent comparison may produce unexpected results. Therefore, use of fractional gate times are discouraged.

Another challenge is presented by setting gate times to zero. Time is used to track dependencies and with zero gate times such approach no longer works. Consider **Figure 1** where each gate has depth zero.

A close up of a whiteboard

Description automatically generated

**Figure 1**

Here user qubit q3 ends at time 0 and user qubit q1 starts at time 0. So user qubit q1 can be assigned to the same underlying qubit after user qubit q3, which is not correct. Even worse, based solely on time, user qubit q1 can be reused as qubit q2, but they are arguments of the same gate, so it is impossible.

One possible solution to this problem is “complex time”, which is described later.

### Gate decomposition

While quantum program may include any gates, underlying quantum device or a target fault-tolerant encoding will not support arbitrary gates. Gates coming from user program need to be decomposed to obtain the sequence of gates that is compatible with the underlying model. Width and depth measurements should be done on this modified sequence of gates. Such decompositions should also be provided.

### Measurements

Measurements directly affect depth calculations and may also affect width indirectly. Consider the following fragment:

if ( Measure( q1 ) ) {  
 X( q2 )  
}

Further suppose that both Measure and X gates take one time step. Without knowing how the outcome of a measurement affects execution path it is not possible to obtain proper depth. This if statement may look like the following sequence to the depth algorithm:

M(q1)  
X(q2)

As these gates operate on different qubits they can be executed in parallel and hence the depth of this circuit is one time step. However X gate cannot be executed at the same time as Measurement in the original program because the outcome of measurement is not known and the depth of the original program is two time steps.

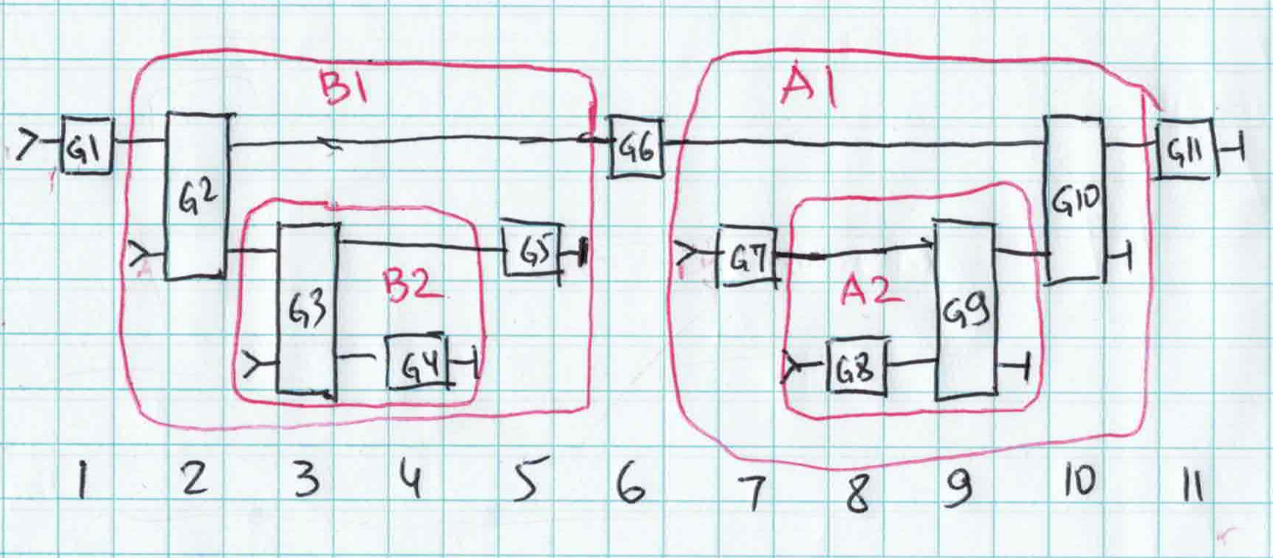
To make this information known to the depth algorithm classical controls may be introduced. This is also outside of the scope of our current implementation. As a result, the depth calculated for a program that uses measurements may be lower than its actual depth.

## Circuit Examples

Several examples are shown to illustrate how to achieve low width and compatible depth as well as minimum depth and compatible width. Most examples are be based on the following Q# program. Here operation names are A1, A2, B1, and B2. Gate names start with G (i.e. G1, G2, G3, etc.) and are numbered in the order they are encountered by the program.

|  |
| --- |
| operation Main(): Unit {  using (q = Qubit()) {  G1(q); B1(q); G6(q); A1(q); G11(q);  }  }  operation A1(q: Qubit): Unit {  using (qq = Qubit()) {  G7(qq); A2(qq); G10(q, qq);  }  }  operation A2(q: Qubit): Unit {  using (qq = Qubit()) {  G8(qq); G9(q, qq);  }  }  operation B1(q: Qubit): Unit {  using (qq = Qubit()) {  G2(q, qq); B2(qq); G5(qq);  }  }  operation B2(q: Qubit): Unit {  using (qq = Qubit()) {  G3(q, qq); G4(qq);  }  } |

The program translates into the circuit shown on :



**Figure 2**

Qubits are horizontal lines, gates are rectangles, operations are circled in red, time goes from left to right. Qubits are maximally reused here. Qubit released after gate G5 is reused for qubit that starts with gate G7. Qubit after gate G4 is reused for qubit starting with gate G8. This gives us the reasonable width, which is also minimal in this case. Depth, without optimizations, is 11. This is not an optimal depth (provided minimum width must be preserved). To obtain optimal depth we schedule gates as early as possible according to the “gravity” algorithm as shown on the :

Diagram

Description automatically generated

**Figure 3**

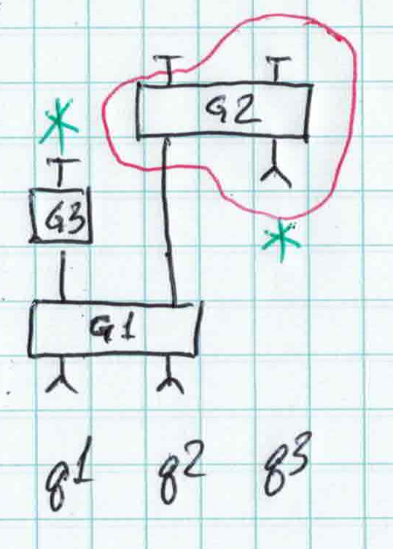
In minimal width of 3 qubits and corresponding optimal depth of 8 is achieved. We can think of this process of applying “gravity” and “shaking” gates. So, it’s more intuitive to draw circuits with time going from top to bottom. The following pictures will use this notation.

`A close up of a whiteboard

Description automatically generated

**Figure 4**

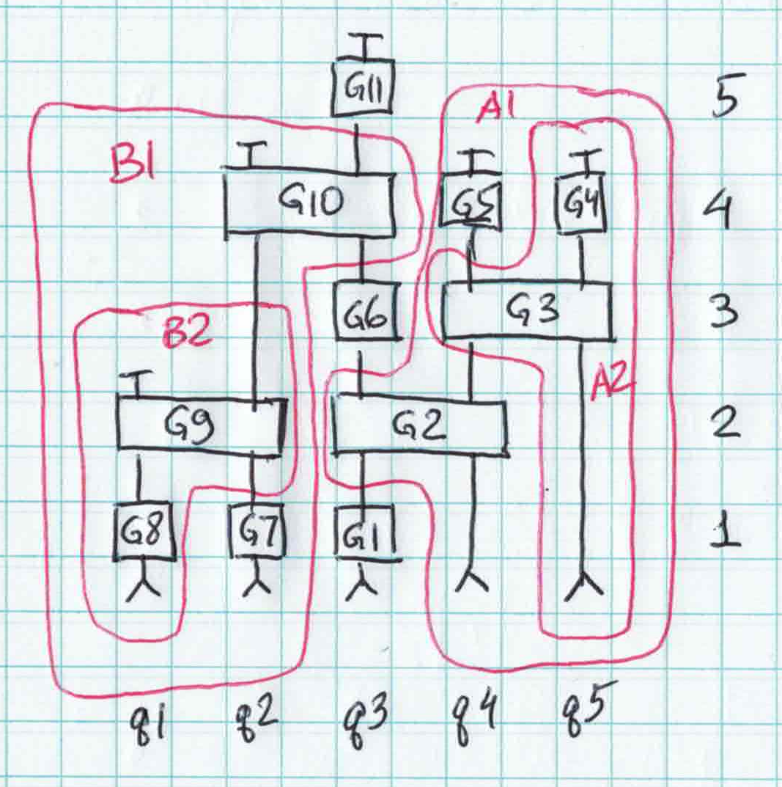
is an example of a circuit built for a different program, which illustrates why our approach don’t result in minimum width. Here no qubits can be reused if we don’t reorder gates. So the width of this circuit will be reported as 3.



**Figure 5**

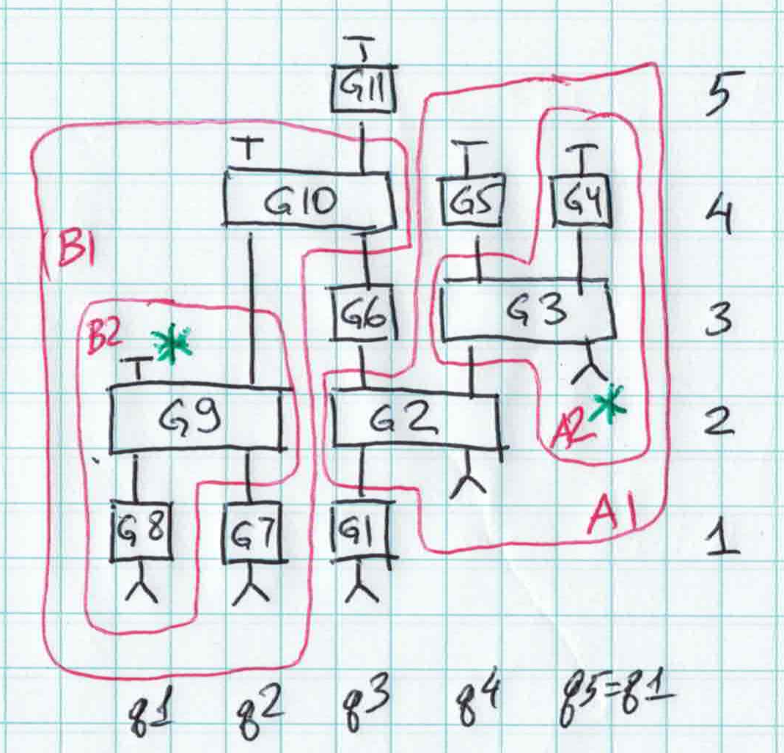
shows equivalent circuit with gates reordered and scheduled at different times. Here qubit q3 can use the same underlying qubit as qubit q1 decreasing width of the circuit to 2. We don’t do this for low width calculations.

What if we want to obtain minimal depth? Qubit reuse may create unnecessary dependencies, so we start without qubit reuse. shows circuit equivalent to without qubit reuse and with gravity applied, which gives us minimum depth of 5.



**Figure 6**

shows minimum depth of 5 and compatible width of 5. This is not an optimal width provided minimum depth should be preserved. Here each qubit is allocated at the very beginning of the circuit. This completely prevents reuse of qubits. We can allocate qubits at the time of the first gate and obtain the circuit shown in (the only difference is that qubits q4 and q5 do not start at 0). Here we can reuse qubit q1 for qubit q5 reducing width from 5 to 4. The green asterisk shows reuse.



**Figure 7**

Please note that when gravity is applied, we schedule gates at different times than they were encountered in Q# program. In gates that are encountered late, such as G8 can be scheduled before gates that are encountered earlier, such as G4. Given constraint to keep the depth at minimum, gates may be rescheduled in many different ways, which affects qubit reuse and hence width. The scheduling problem is computationally hard problem so we cannot hope for the optimal solution here, we can only hope for a reasonable solution for common cases.

Qubit reuse here is also out-of-order: we assign both qubit q1 and qubit q5 to the same underlying qubit, with qubit q1 *before* qubit q5 on timeline. However, the program encounters qubit q5 before qubit q1. Such scenarios are supported by reuse algorithm.

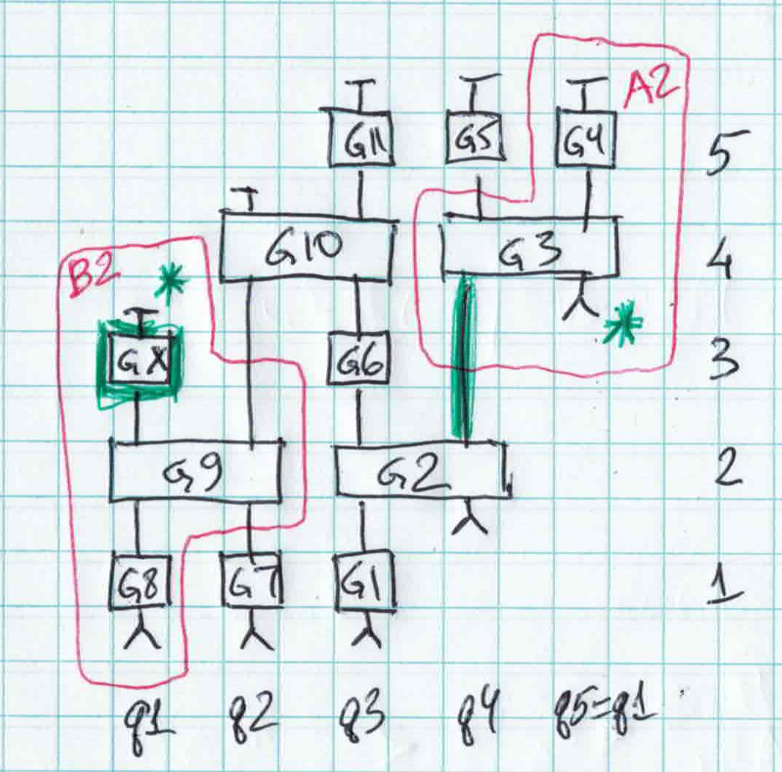
shows use of several heuristics: applying gravity to schedule gates as early as possible, allocating qubit right before the first gate that uses it, and reuse of qubits based on busy time. Not all heuristics work well. One might think that scheduling in such a way that minimizes idle time between gates would be best. shows such scheduling with no idle time between the gates, yet no qubits can be reused in this case. This example show that width optimization (given minimal depth) is a global optimization problem rather than local – situation encountered in one part of the circuit may affect a very distant part of it.

Diagram, engineering drawing

Description automatically generated

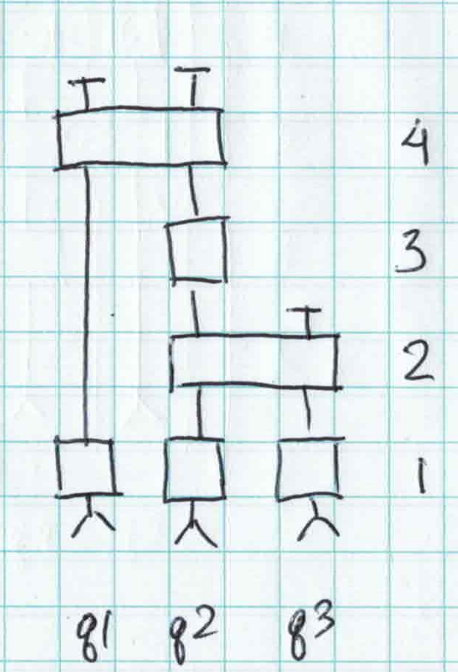
**Figure 8**

shows a slightly modified circuit with gate GX added (highlighted in green). It illustrates that the gravity heuristic does not always produce optimal results. Here the q4 qubit idle time between gates G2 and G3 must be extended (highlighted in green) to reuse qubit q1 for q5. If gravity was applied, G3 would be scheduled right after G2 and reuse would not be possible. Adding idle time does not increase overall depth but allows for qubit reuse and hence produces lower width.



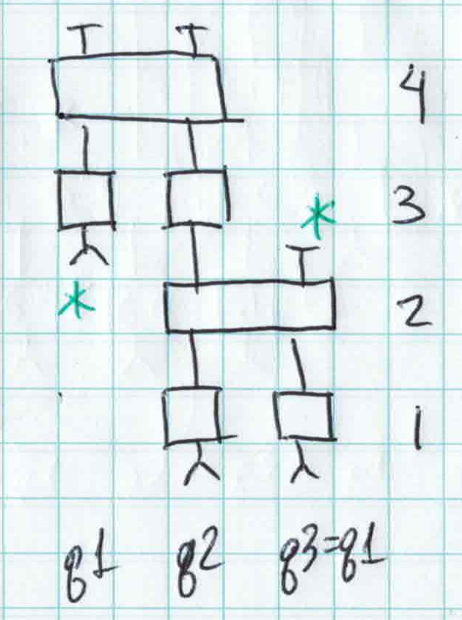
**Figure 9**

shows another common case where a single-qubit gate is applied to the newly allocated qubit q1. By applying gravity, this gate is scheduled at the very beginning of the circuit.



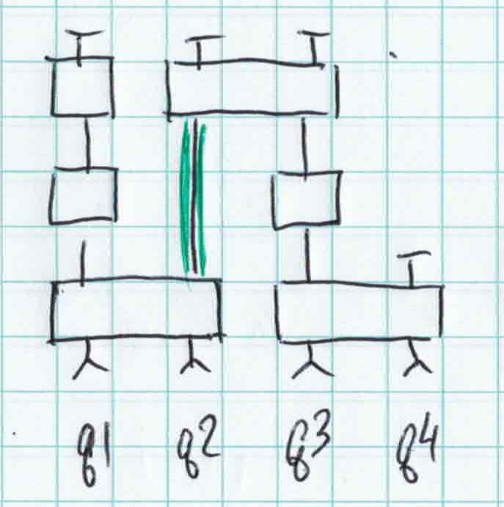
**Figure 10**

Therefore in qubits cannot be reused. This can be rectified by moving first gate on qubit q1 up. This is shown on .



**Figure 11**

allows for reuse of qubits by brining a string of single-qubit gates (only one in this case) on a freshly allocated up to a multi-qubit gate. Such heuristic is safe – it cannot increase circuit depth or prevent reuse of qubits. On the other hand, bringing two-qubit gate up is not safe. This can be seen on .



**Figure 12**

shows the circuit where brining up two-qubit gate on qubits q1 and q2 may enable reuse of qubits but will increase overall circuit depth.

## Current approach to minimal depth

* We associate two depth values with every user qubit – start time and end time. Qubits are considered busy from start time to end time and free outside this interval. On allocations, end time is initialized with zero. Start time is initialized with undefined value indicating that the qubit is not fixed in time.
* If a qubit is not fixed in time and a single qubit gate is encountered, the end time is advanced by the depth of the gate.
* If a multi-qubit gate is encountered, its start time is calculated as maximum of end times of all its arguments. Then all argument qubits are fixed in time by calculating start time for all arguments that aren’t yet fixed in time. Then end time for all arguments are updated. This is the same approach as “gravity” with addition of calculating start times. Such calculation corresponds to bringing a string of single-qubit gates up.
* When user qubit is released, if it is not yet fixed in time, its start time is set to zero. By doing this we schedule qubits with only single-qubit gates to start at zero.
* Then reuse possibilities are checked. If there are underlying qubits with start time after the end time of this user qubit or if there are underlying qubits with end time before the start time of this user qubit then they are considered as candidates for reuse.
* If there are no candidates for reuse, then a new underlying qubit is allocated, and its start and end times are copied from the user qubit.
* If there are candidates for reuse among existing underlying qubits, one with the smallest gap is selected. It’s start time or end time is updated to include the user qubit interval.
* This approach will not work if gate times are set to zero as dependencies would not be properly tracked. To overcome this problem a “complex” time consisting of two components is used. First component tracks actual depth as specified by the user. The second component counts gates when actual depth is zero.
* The first component of complex time has precedence in comparisons. When complex time needs to be advanced by the gate depth, the first component is advanced when the gate depth is non-zero and the second component is reset. When the gate depth is zero, second component is increased by one.
* Maximum of all end times of all underlying qubits is reported as depth and the number of underlying qubits is reported as width.