

Guide for the Quill Package

v0.2.1 March 11, 2024

<https://github.com/Mc-Zen/quill>

Mc-Zen

Quill is a library for creating quantum circuit diagrams in [Typst](#).

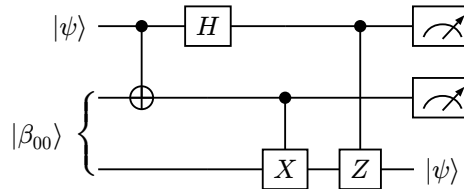
CONTENTS

I Introduction	2
II Basics	2
II.a Multi-Qubit Gates and Wire Labels	3
II.b All about Wires	4
II.c Slices and Gate Groups	5
II.d Labels	5
III Circuit Styling	7
IV Gate Gallery	9
V Fine-Tuning	10
VI Annotations	11
VII Custom Gates	12
VIII Function Documentation	13
IX Demo	22
IX.a Quantum Teleportation	22
IX.b Quantum Phase Estimation	22
IX.c Quantum Fourier Transform:	23
IX.d Shor Nine Qubit Code	23
IX.e Fault-Tolerant Measurement	24
IX.f Fault-Tolerant Gate Construction	24
Bibliography	25

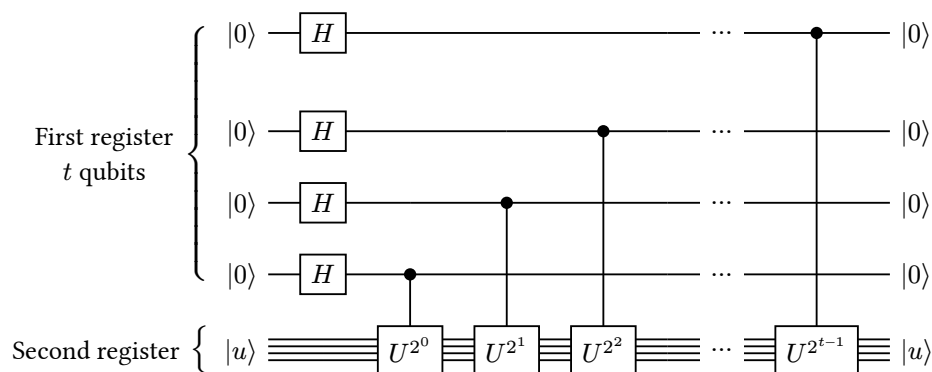
I INTRODUCTION

Section IV features a gallery of many gates and symbols and how to create them. In Section IX, you can find a variety of example figures along with the code.

Would you like to create quantum circuits directly in Typst? Maybe a circuit for quantum teleportation?



Or one for phase estimation? The code for both examples can be found in Section IX.



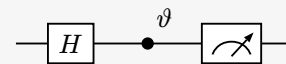
This library provides high-level functionality for generating these and more quantum circuit diagrams.

For those who work with the LaTeX packages `qcircuit` and `quantikz`, the syntax will be familiar. The wonderful thing about Typst is that the changes can be viewed instantaneously which makes it ever so much easier to design a beautiful quantum circuit. The syntax also has been updated a little bit to fit with concepts of the Typst language and many things like styling content is much simpler than with `quantikz` since it is directly supported in Typst.

II BASICS

A circuit can be created by calling the `quantum-circuit()` function with a number of circuit elements.

```
#quantum-circuit(
  1, gate($H$), phase($\theta$), meter(), 1
)
```



A quantum gate is created with the `gate()` command. To make life easier, instead of calling `gate(H)`, you can also just put in the gate's content `H`. Unlike `qcircuit` and `quantikz`, the math environment is not automatically entered for the content of the gate which allows for passing in any type of content (even images or tables). Use `displaystyle math` (for example `$ u_1 $` instead of `$u_1$` to enable appropriate scaling of the gate for more complex mathematical expressions like double subscripts etc.

Consecutive gates are automatically joined with wires. Plain integers can be used to indicate a number of cells with just wire and no gate (where you would use a lot of `&`'s and `\qw`'s in quantikz).

```
#quantum-circuit(
  1, $H$, 4, meter()
)
```



A new wire can be created by breaking the current wire with `[\]`:

```
#quantum-circuit(
  1, $H$, ctrl(1), 1, [ \ ],
  2, targ(), 1
)
```

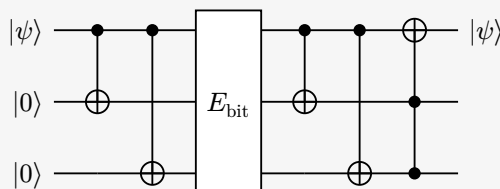


We can create a cx-gate by calling `ctrl()` and passing the relative distance to the desired wire, e.g., `1` to the next wire, `2` to the second-next one or `-1` to the previous wire. Per default, the end of the vertical wire is just joined with the target wire without any decoration at all. Here, we make the gate a cx-gate by adding a `targ()` symbol on the second wire. In order to make a cz-gate with another control circle on the target wire, just use `ctrl(0)` as target.

II.a Multi-Qubit Gates and Wire Labels

Let's look at a quantum bit-flipping error correction circuit. Here we encounter our first multi-qubit gate as well as wire labels:

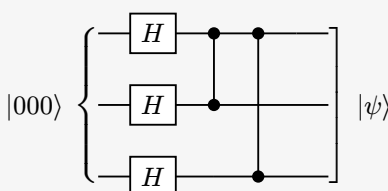
```
#quantum-circuit(
  lstick($|psi>), ctrl(1), ctrl(2), mqgate($E_"bit"$, n: 3), ctrl(1), ctrl(2),
  targ(), rstick($|psi>), [ \ ],
  lstick($|0>), targ(), 2, targ(), 1, ctrl(-1), 1, [ \ ],
  lstick($|0>), 1, targ(), 2, targ(), ctrl(-1), 1
)
```



Multi-qubit gates have a dedicated command `mqgate()` which allows to specify the number of qubits `n` as well as a variety of other options. Wires can be labelled at the beginning or the end with the `lstick()` and `rstick()` commands, respectively. Both create a label “sticking” out from the wire.

Just as multi-qubit gates, `lstick()` and `rstick()` can span multiple wires, again with the parameter `n`. Furthermore, the brace can be changed or turned off with `brace: none`. If the label is only applied to a single qubit, it will have no brace by default but in this case a brace can be added just the same way. By default it is set to `brace: auto`.

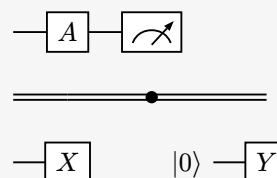
```
#quantum-circuit(
  lstick($|000>$, n: 3), $H$, ctrl(1), ctrl(2), 1,
  rstick($|psi>$, n: 3, brace: "]"), [\ ],
  1, $H$, ctrl(0), 3, [\ ],
  1, $H$, 1, ctrl(0), 2
)
```



II.b All about Wires

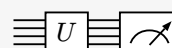
In many circuits, we need classical wires. This library generalizes the concept of quantum, classical and bundled wires and provides the `setwire()` command that allows all sorts of changes to the current wire setting. You may call `setwire()` with the number of wires to display:

```
#quantum-circuit(
  1, $A$, meter(n: 1), [\ ],
  setwire(2), 2, ctrl(0), 2, [\ ],
  1, $X$, setwire(0), 1, lstick($|0>$), setwire(1), $Y$,
)
```



The `setwire()` command produces no cells and can be called at any point on the wire. When a new wire is started, the default wire setting is restored automatically (quantum wire with default wire style, see Section III on how to customize the default). Calling `setwire(0)` removes the wire altogether until `setwire()` is called with different arguments. More than two wires are possible and it lies in your hands to decide how many wires still look good. The distance between wires can also be specified:

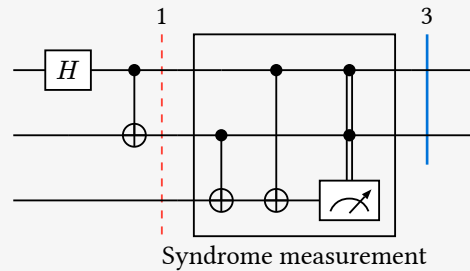
```
#quantum-circuit(
  setwire(4, wire-distance: 1.5pt), 1, $U$, meter()
)
```



II.c Slices and Gate Groups

In order to structure quantum circuits, you often want to mark sections to denote certain steps in the circuit. This can be easily achieved through the `slice()` and `gategroup()` commands. Both are inserted into the circuit where the slice or group should begin and allow an arbitrary number of labels through the `labels` argument (more on labels in Section II.d). The function `gategroup()` takes two positional integer arguments which specify the number of wires and steps the group should span. Slices reach down to the last wire by default but the number of sliced wires can also be set manually.

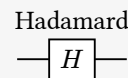
```
#quantum-circuit(
  1, gate($H$), ctrl(1),
  slice(label: "1"), 1,
  gategroup(3, 3, label: (content:
    "Syndrome measurement", pos: bottom)),
  1, ctrl(2), ctrl(0), 1,
  slice(label: "3", n: 2,
    stroke: blue),
  2, [ \ ],
  2, targ(), 1, ctrl(1), 1, ctrl(0), 3, [ \ ],
  4, targ(), targ(), meter(target: -2)
)
```



II.d Labels

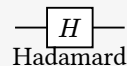
Finally, we want to show how to place labels on gates and vertical wires. The function `gate()` and all the derived gate commands such as `meter()`, `ctrl()`, `lstick()` etc. feature a `label` argument for adding any number of labels on and around the element. In order to produce a simple label on the default position (for plain gates this is at the top of the gate, for vertical wires it is to the right and for the `phase()` gate it is to the top right), you can just pass content or a string:

```
#quantum-circuit(
  1, gate($H$, label: "Hadamard"), 1
)
```



If you want to change the position of the label or specify the offset, you want to pass a dictionary with the key `content` and optional values for `pos` (alignment), `dx` and `dy` (length, ratio or relative length):

```
#quantum-circuit(
  1, gate($H$, label: (content: "Hadamard", pos: bottom, dy: 0pt)), 1
)
```

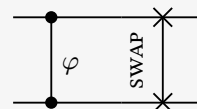


Multiple labels can be added by passing an array of labels specified through dictionaries.

```
#quantum-circuit(
  1, gate(hide($H$), label: (
    (content: "lt", pos: left + top),
    (content: "t", pos: top),
    (content: "rt", pos: right + top),
    (content: "l", pos: left),
    (content: "c", pos: center),
    (content: "r", pos: right),
    (content: "lb", pos: left + bottom),
    (content: "b", pos: bottom),
    (content: "rb", pos: right + bottom),
  )), 1
)
```

Labels for slices and gate groups work just the same. In order to place a label on a control wire, you can use the wire-label parameter provided for [mqgate\(\)](#), [ctrl\(\)](#) and [swap\(\)](#).

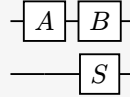
```
#quantum-circuit(
  1, ctrl(1, wire-label: $phi$), 2,
  swap(1, wire-label: (
    content: rotate(-90deg, smallcaps("swap")),
    pos: left, dx: 0pt)
  ), 1, [\ ], 10pt,
  1, ctrl(0), 2, swap(0), 1,
)
```



III CIRCUIT STYLING

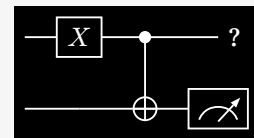
The `quantum-circuit()` command provides several options for styling the entire circuit. The parameters `row-spacing` and `column-spacing` allow changing the optical density of the circuit by adjusting the spacing between circuit elements vertically and horizontally.

```
#quantum-circuit(  
  row-spacing: 5pt,  
  column-spacing: 5pt,  
  1, $A$, $B$, 1, [\ ],  
  1, 1, $S$, 1  
)
```



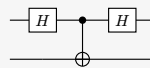
The `wire`, `color` and `fill` options provide means to customize line strokes and colors. This allows us to easily create “dark-mode” circuits:

```
#box(fill: black, quantum-circuit(  
  wire: .7pt + white, // Wire and stroke color  
  color: white,        // Default foreground and text color  
  fill: black,         // Gate fill color  
  1, $X$, ctrl(1), rstick([*?*]), [\ ],  
  1,1, targ(), meter(),  
))
```



Furthermore, a common task is changing the total size of a circuit by scaling it up or down. Instead of tweaking all the parameters like font-size, padding, row-spacing etc. you can specify the `scale` option which takes a percentage value:

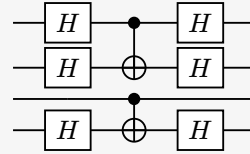
```
#quantum-circuit(  
  scale: 60%,  
  1, $H$, ctrl(1), $H$, 1, [\ ],  
  1, 1, targ(), 2  
)
```



Note, that this is different than calling Typst’s built-in `scale()` function on the circuit which would scale it without affecting the layout, thus still reserving the same space as if unscaled!

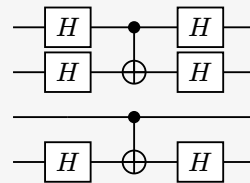
For an optimally layout, the height for each row is determined by the gates on that wire. For this reason, the wires can have different distances. To better see the effect, let's decrease the row-spacing:

```
#quantum-circuit(
    row-spacing: 2pt, min-row-height: 4pt,
    1, $H$, ctrl(1), $H$, 1, [\ ],
    1, $H$, targ(), $H$, 1, [\ ],
    2, ctrl(1), 2, [\ ],
    1, $H$, targ(), $H$, 1
)
```



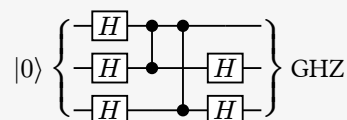
Setting the option `equal-row-heights` to `true` solves this problem (manually spacing the wires with lengths is still possible, see Section V):

```
#quantum-circuit(
    equal-row-heights: true,
    row-spacing: 2pt, min-row-height: 4pt,
    1, $H$, ctrl(1), $H$, 1, [\ ],
    1, $H$, targ(), $H$, 1, [\ ],
    2, ctrl(1), 2, [\ ],
    1, $H$, targ(), $H$, 1
)
```



There is another option for `quantum-circuit()` that has a lot of impact on the looks of the diagram: `gate-padding`. This at the same time controls the default gate box padding and the distance of `lstick` and `rstick` to the wire. Need really wide or tight circuits?

```
#quantum-circuit(
    gate-padding: 2pt,
    row-spacing: 5pt, column-spacing: 7pt,
    lstick($|0\rangle$, n: 3), $H$, ctrl(1),
    ctrl(2), 1, rstick("GHZ", n: 3), [\ ],
    1, $H$, ctrl(0), 1, $H$, 1, [\ ],
    1, $H$, 1, ctrl(0), $H$, 1
)
```



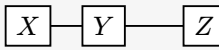
IV GATE GALLERY

Normal gate		<code>gate(\$H\$), \$H\$</code>	Round gate		<code>gate(\$X\$, radius: 100%)</code>
D gate		<code>gate(\$Y\$, radius: (right: 100%))</code>	Meter		<code>meter()</code>
Meter with label		<code>meter(label: \$lr(±>)\$)</code>	Phase gate		<code>phase(\$α\$)</code>
Control		<code>ctrl(0)</code>	Open control		<code>ctrl(0, open: true)</code>
Target		<code>targ()</code>	Swap target		<code>swap(0)</code>
Permutation gate		<code>permute(2,0,1)</code>	Multiqubit gate		<code>mqqgate(\$U\$, 3)</code>
lstick		<code>lstick(\$ psi\$)</code>	rstick		<code>rstick(\$ psi\$)</code>
Multi-qubit lstick		<code>lstick(\$ psi\$,\$, n: 2)</code>	Multi-qubit rstick		<code>rstick(\$ psi\$,\$, n: 2, brace: "]")</code>
midstick		<code>midstick("yeah")</code>	Wire bundle		<code>nwire(5)</code>
Controlled z-gate		<code>ctrl(1) + ctrl(0)</code>	Controlled x-gate		<code>ctrl(1) + targ()</code>
Swap gate		<code>swap(1) + targX()</code>	Controlled Hadamard		<code>mqqgate(\$H\$,target: 1) + ctrl(0)</code>
Plain vertical wire		<code>ctrl(1, show-dot: false)</code>	Meter to classical		<code>meter(target: 1) + ctrl(0)</code>
Classical wire		<code>setwire(2)</code>	Styled wire		<code>setwire(1, stroke: green)</code>
Labels		<pre>gate(\$Q\$, label: ((content: "b",pos:top), (content:"b",pos:bottom), (content: "a", pos: left + top), (content: "c", pos: right + top, dy: 0pt, dx: 50%),))</pre>	Gate inputs and outputs		<pre>mqqgate(\$U\$, n: 3, width: 5em, inputs: ((qubit:0, n:2, label:\$x\$), (qubit:2, label: \$y\$)), outputs: ((qubit:0, n:2, label:\$x\$), (qubit:2, label:\$y\oplus f(x)\$)))</pre>

V FINE-TUNING

The `quantum-circuit()` command allows not only gates as well as content and string items but only length parameters which can be used to tweak the appearance of the circuit. Inserting a length value between gates adds a **horizontal space** of that length between the cells:

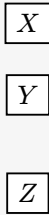
```
#quantum-circuit(  
  $X$, $Y$, 10pt, $Z$  
)
```



In the background, this works like a grid gutter that is set to `0pt` by default. If a length value is inserted between the same two columns on different wires/rows, the maximum value is used for the space. In the same spirit, inserting multiple consecutive length values result in the largest being used, e.g., a `5pt, 10pt, 6pt` results in a `10pt` gutter in the corresponding position.

Putting a length after the wire break item `[\]` produces a **vertical space** between the corresponding wires:

```
#quantum-circuit(  
  $X$, [\ ], $Y$, [\ ], 10pt, $Z$  
)
```



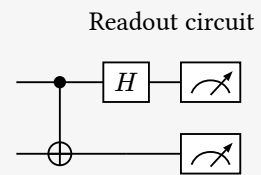
VI ANNOTATIONS

Quill provides a way of making custom annotations through the `annotate()` interface. An `annotate()` object may be placed anywhere in the circuit, the position only matters for the draw order in case several annotations would overlap.

The `annotate()` command allows for querying cell coordinates of the circuit and passing in a custom draw function to draw globally in the circuit diagram.

Let's look at an example:

```
#quantum-circuit(
  1, ctrl(1), $H$, meter(), [\ ],
  1, targ(), 1, meter(),
  annotate((2, 4), 0, ((x1, x2), y) => {
    let brace = math.ln($#block(height: x2 - x1)$)
    place(dx: x1, dy: y, rotate(brace, -90deg, origin: top))
    let content = [Readout circuit]
    style(styles => {
      let size = measure(content, styles)
      place(
        dx: x1 + (x2 - x1) / 2 - size.width / 2,
        dy: y - .6em - size.height, content
      )
    })
  })
)
```

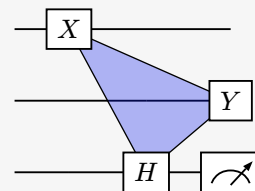


First, the call to `annotate()` asks for the x coordinates of the second and forth column and the y coordinate of the zeroth row (first wire). The draw callback function then gets the corresponding coordinates as arguments and uses them to draw a brace and some text above the cells.

Note, that the circuit does not know how large the annotation is. If it goes beyond the circuits bounds, you may want to adjust the parameter `circuit-padding` of `quantum-circuit()` appropriately.

Another example, here we want to obtain coordinates for the cell centers. We can achieve this by adding 0.5 to the cell index. The fractional part of the number represents a percentage of the cell width/height.

```
#quantum-circuit(
  1, $X$, 2, [\ ],
  1, 2, $Y$, [\ ],
  1, 1, $H$, meter(),
  annotate((1.5, 3.5, 2.5), (0.5, 1.5, 2.5),
    ((x0, x1, x2), (y0, y1, y2)) => {
      path(
        (x0, y0), (x1, y1), (x2, y2),
        closed: true,
        fill: rgb("#1020EE50"), stroke: .5pt + black
      )
    })
)
```



VII CUSTOM GATES

Quill allows you to create totally customized gates by specifying the draw-function argument in `gate()` or `mqgate()`. You will not need to do this however if you just want to change the color of the gate or make it round. For these tasks you can just use the appropriate arguments of the `gate()` command.

Note, that the interface for custom gates might still change a bit.


When the circuit is layed out, the draw function is called with two (read-only) arguments: the gate itself and a dictionary that contains information about the circuit style and more.

Let us look at a little example for a custom gate that just shows the vertical lines of the box but not the horizontal ones.

```
#let draw-quill-gate(gate, draw-params) = {  
  let stroke = draw-params.wire  
  let fill = if gate.fill != none { gate.fill } else { draw-params.background }  
  
  box(  
    gate.content,  
    fill: fill, stroke: (left: stroke, right: stroke),  
    inset: draw-params.padding  
  )  
}
```

We can now use it like this:

```
#quantum-circuit(  
  1, gate("Quill", draw-function: draw-quill-gate), 1,  
)
```



The first argument for the draw function contains information about the gate. From that we read the gate's content (here "Quill"). We create a `box()` with the content and only specify the left and right edge stroke. In order for the circuit to look consistent, we read the circuit style from the draw-params. The key `draw-params.wire` contains the (per-circuit) global wire stroke setting as set through `quantum-circuit(wire: ...)`. Additionally, if a fill color has been specified for the gate, we want to use it. Otherwise, we use `draw-params.background` to be conform with for example dark-mode circuits. Finally, to create space, we add some inset to the box. The key `draw-params.padding` holds the (per-circuit) global gate padding length.

It is generally possible to read any value from a gate that has been provided in the gate's constructor. Currently, content, fill, radius, width, box and data (containing the optional data argument that can be added in the `gate()` function) can be read from the gate. For multi-qubit gates, the key `multi` contains a dictionary with the keys `target` (specifying the relative target qubit for control wires), `num-qubits`, `wire-count` (the wire count for the control wire) and `extent` (the amount of length to extend above the first and below the last wire).

All built-in gates are drawn with a dedicated draw-function and you can also take a look at the source code for ideas and hints.

VIII FUNCTION DOCUMENTATION

This section contains a complete reference for every function in **quill**.

Quantum Circuit

- [quantum-circuit\(\)](#)

Gates

- [gate\(\)](#)
- [mqgate\(\)](#)
- [meter\(\)](#)
- [permute\(\)](#)
- [phantom\(\)](#)
- [targ\(\)](#)
- [targX\(\)](#)
- [phase\(\)](#)
- [swap\(\)](#)
- [ctrl\(\)](#)

Decorations

- [lstick\(\)](#)
- [rstick\(\)](#)
- [midstick\(\)](#)
- [nwire\(\)](#)
- [setwire\(\)](#)
- [gategroup\(\)](#)
- [slice\(\)](#)
- [annotate\(\)](#)

quantum-circuit

Create a quantum circuit diagram. Children may be

- gates created by one of the many gate commands ([gate\(\)](#), [mqgate\(\)](#), [meter\(\)](#), ...),
- `[\]` for creating a new wire/row,
- commands like [setwire\(\)](#), [slice\(\)](#) or [gategroup\(\)](#),
- integers for creating cells filled with the current wire setting,
- lengths for creating space between rows or columns,
- plain content or strings to be placed on the wire, and
- [lstick\(\)](#), [midstick\(\)](#) or [rstick\(\)](#) for placement next to the wire.

Parameters

```
quantum-circuit(  
  wire: stroke ,  
  row-spacing: length ,  
  column-spacing: length ,  
  min-row-height: length ,  
  min-column-width: length ,  
  gate-padding: length ,  
  equal-row-heights: boolean ,  
  color: color ,  
  fill: color ,  
  font-size: length ,  
  scale: ratio ,  
  scale-factor,  
  baseline: length content string ,  
  circuit-padding: dictionary ,  
  ..children: array  
)
```

wire stroke

Style for drawing the circuit wires. This can take anything that is valid for the stroke of the builtin [line\(\)](#) function.

Default: `.7pt + black`

row-spacing length

Spacing between rows.

Default: `12pt`

column-spacing length

Spacing between columns.

Default: `12pt`

min-row-height length

Minimum height of a row (e.g., when no gates are given).

Default: `10pt`

min-column-width length

Minimum width of a column.

Default: `0pt`

gate-padding length

General padding setting including the inset for gate boxes and the distance of [lstick\(\)](#) and co. to the wire.

Default: `.4em`

equal-row-heights boolean

If true, then all rows will have the same height and the wires will have equal distances orienting on the highest row.

Default: `false`

color color

Foreground color, default for strokes, text, controls etc. If you want to have dark-themed circuits, set this to white for instance and update `wire` and `fill` accordingly.

Default: `black`

fill `color`

Default fill color for gates.

Default: white

font-size `length`

Default font size for text in the circuit.

Default: 10pt

scale `ratio`

Total scale factor applied to the entire circuit without changing proportions

Default: 100%

baseline `length` or `content` or `string`

Set the baseline for the circuit. If a content or a string is given, the baseline will be adjusted automatically to align with the center of it. One useful application is `"="` so the circuit aligns with the equals symbol.

Default: 0pt

circuit-padding `dictionary`

Padding for the circuit (e.g., to accommodate for annotations) in form of a dictionary with possible keys `left`, `right`, `top` and `bottom`. Not all of those need to be specified.

This setting basically just changes the size of the bounding box for the circuit and can be used to increase it when labels or annotations extend beyond the actual circuit.

Default: .4em

..children `array`

Items, gates and circuit commands (see description).

ctrl

Creates a control with a vertical wire to another qubit.

Parameters

```
ctrl(  
  n: integer,  
  wire-count: integer,  
  open: boolean,  
  fill: none or color,  
  size: length,  
  show-dot: boolean,  
  label,  
  wire-label: array or string or content or dictionary  
)
```

n `integer`

How many wires up or down the target wire lives.

wire-count `integer`

Wire count for the control wire.

Default: 1

open `boolean`

Whether to draw an open dot.

Default: false

fill `none` or `color`

Fill color for the circle or stroke color if `open: true`.

Default: none

size `length`

Size of the control circle.

Default: 2.3pt

show-dot `boolean`

Whether to show the control dot. Set this to false to obtain a vertical wire with no dots at all.

Default: true

wire-label `array` or `string` or `content` or `dictionary`

One or more labels to add to the control wire. See [mqgate\(\)](#).

Default: none

gate

This is the basic command for creating gates. Use this to create a simple gate, e.g., `gate(X)`. For special gates, many other dedicated gate commands exist.

Note, that most of the parameters listed here are mostly used for derived gate functions and do not need to be touched in all but very few cases.

Parameters

```
gate(  
  content: content,  
  fill: none or color,  
  radius: length or dictionary,  
  width: auto or length,  
  box: boolean,  
  floating: boolean,  
  multi: dictionary,  
  size-hint: function,  
  draw-function: function,  
  gate-type,  
  data: any,  
  label: array or string or content or dictionary  
)
```

content `content`

What to show in the gate (may be none for special gates like `ctrl()`).

fill `none` or `color`

Gate background fill color.

Default: `none`

radius `length` or `dictionary`

Gate rectangle border radius. Allows the same values as the builtin `rect()` function.

Default: `0pt`

width `auto` or `length`

The width of the gate can be specified manually with this property.

Default: `auto`

box `boolean`

Whether this is a boxed gate (determines whether the outgoing wire will be drawn all through the gate (`box: false`) or not).

Default: `true`

floating `boolean`

Whether the content for this gate will be shown floating (i.e. no width is reserved).

Default: `false`

multi `dictionary`

Information for multi-qubit and controlled gates (see `mqgate()`).

Default: `none`

size-hint `function`

Size hint function. This function should return a dictionary containing the keys `width` and `height`. The result is used to determine the gates position and cell sizes of the grid. Signature: `(gate, draw-params) => {}`.

Default: `layout.default-size-hint`

draw-function `function`

Drawing function that produces the displayed content. Signature: `(gate, draw-params) => {}`.

Default: `draw-functions.draw-boxed-gate`

data `any`

Optional additional gate data. This can for example be a dictionary storing extra information that may be used for instance in a custom `draw-function`.

Default: `none`

label `array` or `string` or `content` or `dictionary`

One or more labels to add to the gate. Usually, a label consists of a dictionary with entries for the keys `content` (the label content), `pos` (2d alignment specifying the position of the label) and optionally `dx` and/or `dy` (lengths, ratios or relative lengths). If only a single label is to be added, a plain content or string value can be passed which is then placed at the default position.

Default: `none`

meter

Draw a meter box representing a measurement.

Parameters

```
meter(
    target: none or integer,
    n: integer,
    wire-count: integer,
    label: array of string content dictionary,
    fill,
    radius
)
```

target none or integer

If given, draw a control wire to the given target qubit the specified number of wires up or down.

Default: **none**

n integer

Number of wires to span this meter across.

Default: **1**

wire-count integer

Wire count for the (optional) control wire.

Default: **2**

label array or string or content or dictionary

One or more labels to add to the gate. See [gate\(\)](#).

Default: **none**

mqgate

Basic command for creating multi-qubit or controlled gates. See also [ctrl\(\)](#) and [swap\(\)](#).

Parameters

```
mqgate(
    content: content,
    n: integer,
    target: none or integer,
    fill: none or color,
    radius: length or dictionary,
    width: auto or length,
    box: boolean,
    wire-count: integer,
    inputs: none or array,
    outputs: none or array,
    extent: auto or length,
    size-all-wires: none or boolean,
    draw-function,
    label: array of string content dictionary,
    wire-label: array of string content dictionary,
    data: any
)
```

content content

n integer

Number of wires the multi-qubit gate spans.

Default: **1**

target none or integer

If specified, a control wire is drawn from the gate up or down this many wires counted from the wire this [mqgate\(\)](#) is placed on.

Default: **none**

fill none or color

Gate background fill color.

Default: **none**

radius length or dictionary

Gate rectangle border radius. Allows the same values as the builtin [rect\(\)](#) function.

Default: **0pt**

width auto or length

The width of the gate can be specified manually with this property.

Default: **auto**

box boolean

Whether this is a boxed gate (determines whether the outgoing wire will be drawn all through the gate (box: **false**) or not).

Default: **true**

wire-count integer

Wire count for control wires.

Default: **1**

inputs `none` or array

You can put labels inside the gate to label the input wires with this argument. It accepts a list of labels, each of which has to be a dictionary with the keys `qubit` (denoting the qubit to label, starting at 0) and `content` (containing the label content). Optionally, providing a value for the key `n` allows for labelling multiple qubits spanning over `n` wires. These are then grouped by a brace.

Default: `none`

outputs `none` or array

Same as `inputs` but for gate outputs.

Default: `none`

extent `auto` or `length`

How much to extent the gate beyond the first and last wire, default is to make it align with an X gate (so `[size of x gate] / 2`).

Default: `auto`

size-all-wires `none` or `boolean`

A single-qubit gate affects the height of the row it is being put on. For multi-qubit gate there are different possible behaviours:

- Affect height on only the first and last wire (`false`)
- Affect the height of all wires (`true`)
- Affect the height on no wire (`none`)

Default: `false`

label array or `string` or `content` or dictionary

One or more labels to add to the gate. See `gate()`.

Default: `none`

wire-label array or `string` or `content` or dictionary

One or more labels to add to the control wire. Works analogous to `labels` but with default positioning to the right of the wire.

Default: `none`

data any

Optional additional gate data. This can for example be a dictionary storing extra information that may be used for instance in a custom `draw-function`.

Default: `none`

permute

Create a visualized permutation gate which maps the qubits q_k, q_{k+1}, \dots to the qubits $q_{p(k)}, q_{p(k+1)}, \dots$ when placed on the qubit k . The permutation map is given by the `qubits` argument. Note, that qubit indices start with 0.

Example:

`permute(1, 0)` when placed on the second wire swaps the second and third wire.

`permute(2, 0, 1)` when placed on wire 0 maps $(0, 1, 2) \mapsto (2, 0, 1)$.

Note also, that the wiring is not very sophisticated and will probably look best for relatively simple permutations. Furthermore, it only works with quantum wires.

Parameters

```
permute(  
    ..qubits: array ,  
    width: length ,  
    bend: ratio ,  
    separation: auto none length color stroke  
)
```

..qubits array

Qubit permutation specification.

width length

Width of the permutation gate.

Default: `30pt`

bend ratio

How much to bend the wires. With `0%`, the wires are straight.

Default: `100%`

separation auto or none or length or color or stroke

Overlapping wires are separated by drawing a thicker line below. With this option, this line can be customized in color or thickness.

Default: `auto`

phantom

Create an invisible (phantom) gate for reserving space. If `content` is provided, the `height` and `width` parameters are ignored and the gate will take the size it would have if `gate(content)` was called.

Instead specifying width and/or height will create a gate with exactly the given size (without padding).

Parameters

```
phantom(  
  content: content,  
  width: length,  
  height: length  
)
```

content `content`

Content to measure for the phantom gate size.

Default: `none`

width `length`

Width of the phantom gate (ignored if `content` is not `none`).

Default: `0pt`

height `length`

Height of the phantom gate (ignored if `content` is not `none`).

Default: `0pt`

phase

Create a phase gate shown as a point on the wire together with a label.

Parameters

```
phase(  
  label: content,  
  open: boolean,  
  fill: none or color,  
  size: length  
)
```

label `content`

Angle value to display.

open `boolean`

Whether to draw an open dot.

Default: `false`

fill `none` or `color`

Fill color for the circle or stroke color if `open: true`.

Default: `none`

size `length`

Size of the circle.

Default: `2.3pt`

swap

Creates a SWAP operation with another qubit.

Parameters

```
swap(  
  n: integer,  
  wire-count,  
  size: length,  
  label,  
  wire-label: array of string or content dictionary  
)
```

n `integer`

How many wires up or down the target wire lives.

size `length`

Size of the target symbol.

Default: `7pt`

wire-label `array` or `string` or `content` or `dictionary`

One or more labels to add to the control wire. See [mqgate\(\)](#).

Default: `none`

targ

Target element for controlled-X operations (\oplus).

Parameters

```
targ(  
  fill: none or color or boolean,  
  size: length,  
  label  
)
```

fill `none` or `color` or `boolean`

Fill color for the target circle. If set to `true`, the target is filled with the circuits background color.

Default: `none`

size `length`

Size of the target symbol.

Default: `4.3pt`

targX

Target element for SWAP operations (×) without vertical wire).

Parameters

```
targX(  
  size: length,  
  label  
)
```

size `length`

Size of the target symbol.

Default: `7pt`

annotate

Lower-level interface to the cell coordinates to create an arbitrary annotation by passing a custom function.

This function is passed the coordinates of the specified cell rows and columns.

Parameters

```
annotate(  
  columns: integer array,  
  rows: integer array,  
  callback: function  
)
```

columns `integer` or `array`

Column indices for which to obtain coordinates.

rows `integer` or `array`

Row indices for which to obtain coordinates.

callback `function`

Function to call with the obtained coordinates. The signature should be with signature `(col-coords, row-coords) => {}`. This function is expected to display the content to draw in absolute coordinates within the circuit.

gategroup

Highlight a group of circuit elements by drawing a rectangular box around them.

Parameters

```
gategroup(  
  wires: integer,  
  steps: integer,  
  padding: length dictionary,  
  stroke: stroke,  
  fill: color,  
  radius: length dictionary,  
  label: array string content dictionary  
)
```

wires `integer`

Number of wires to include.

steps `integer`

Number of columns to include.

padding `length` or `dictionary`

Padding of rectangle. May be one length for all sides or a dictionary with the keys `left`, `right`, `top`, `bottom` and `default`. Not all keys need to be specified. The value for `default` is used for the omitted sides or `0pt` if no `default` is given.

Default: `0pt`

stroke `stroke`

Stroke for rectangle.

Default: `.7pt`

fill `color`

Fill color for rectangle.

Default: `none`

radius length or dictionary

Corner radius for rectangle.

Default: 0pt

label array or string or content or dictionary

One or more labels to add to the group. See [gate\(\)](#).

Default: none

lstick

Basic command for labelling a wire at the start.

Parameters

```
lstick(  
  content: content,  
  n: content,  
  brace: auto none string,  
  label: array string content dictionary  
)
```

content content

Label to display, e.g., $|0\rangle$.

n content

How many wires the `lstick` should span.

Default: 1

brace auto or none or string

If `brace` is `auto`, then a default `{` brace is shown only if `n > 1`. A brace is always shown when explicitly given, e.g., `"}"`, `"["` or `"|"`. No brace is shown for `brace: none`

Default: auto

label array or string or content or dictionary

One or more labels to add to the gate. See [gate\(\)](#).

Default: none

midstick

Create a midstick, i.e., a mid-circuit text.

Parameters

```
midstick(  
  content: content,  
  fill,  
  label: array string content dictionary  
)
```

content content

Label to display, e.g., $|0\rangle$.

label array or string or content or dictionary

One or more labels to add to the gate.

Default: none

nwire

Creates a symbol similar to `\qwbundle` on `quantikz`. Annotates a wire to be a bundle of quantum or classical wires.

Parameters

```
nwire(label: integer content)
```

label integer or content

rstick

Basic command for labelling a wire at the end.

Parameters

```
rstick(  
  content: content,  
  n: content,  
  brace: auto none string,  
  label: array string content dictionary  
)
```

content content

Label to display, e.g., $|0\rangle$.

n content

How many wires the `rstick` should span.

Default: 1

brace `auto` or `none` or `string`

If `brace` is `auto`, then a default `}` brace is shown only if `n > 1`. A brace is always shown when explicitly given, e.g., `"}"`, `"["` or `"|"`. No brace is shown for `brace: none`.

Default: `auto`

label `array` or `string` or `content` or dictionary

One or more labels to add to the gate. See [gate\(\)](#).

Default: `none`

n `integer`

Number of wires to slice.

Default: `0`

stroke `stroke`

Line style for the slice.

Default: (paint: `red`, thickness: `.7pt`, dash: `"dashed"`)

label `array` or `string` or `content` or dictionary

One or more labels to add to the slice. See [gate\(\)](#).

Default: `none`

setwire

Set current wire mode (0: none, 1 wire: quantum, 2 wires: classical, more are possible) and optionally the stroke style.

The wire style is reset for each row.

Parameters

```
setwire(  
    wire-count: integer,  
    stroke: none stroke,  
    wire-distance: length  
)
```

wire-count `integer`

Number of wires to display.

stroke `none` or `stroke`

When given, the stroke is applied to the wire. Otherwise the current stroke is kept.

Default: `none`

wire-distance `length`

Distance between wires.

Default: `1pt`

slice

Slice the circuit vertically, showing a separation line between columns.

Parameters

```
slice(  
    n: integer,  
    stroke: stroke,  
    label: array string content dictionary  
)
```

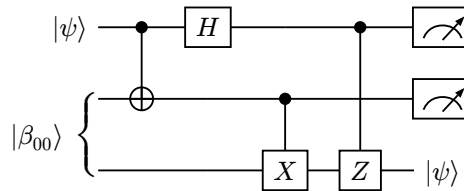
IX DEMO

This section demonstrates the use of the **quantum-circuit** library by reproducing some figures from the famous book *Quantum Computation and Quantum Information* by Nielsen and Chuang [1].

IX.a Quantum Teleportation

Quantum teleportation circuit reproducing the Figure 4.15 in [1].

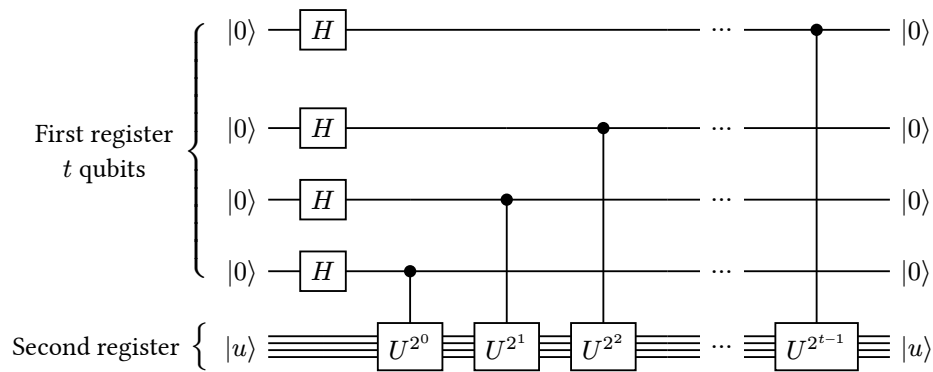
```
#quantum-circuit(
    lstick($|psi>$), ctrl(1), gate($H$), 1, ctrl(2), meter(), [\ ],
    lstick($|beta_00>$, n: 2), targ(), 1, ctrl(1), 1, meter(), [\ ],
    3, gate($X$), gate($Z$), midstick($|psi>$)
)
```



IX.b Quantum Phase Estimation

Quantum phase estimation circuit reproducing the Figure 5.2 in [1].

```
#quantum-circuit(
    setwire(0), lstick(align(center)[First register\ $t$ qubits], n: 4), lstick($|0>$),
    setwire(1), $H$, 4, midstick($ dots $), ctrl(4), rstick($|0>$), [\ ], 10pt,
    setwire(0), phantom(width: 13pt), lstick($|0>$), setwire(1), $H$, 2, ctrl(3), 1,
    midstick($ dots $), 1, rstick($|0>$), [\ ],
    setwire(0), 1, lstick($|0>$), setwire(1), $H$, 1, ctrl(2), 2,
    midstick($ dots $), 1, rstick($|0>$), [\ ],
    setwire(0), 1, lstick($|0>$), setwire(1), $H$, ctrl(1), 3, midstick($ dots $), 1,
    rstick($|0>$), [\ ],
    setwire(0), lstick([Second register], n: 1, brace: "{"), lstick($|u>$),
    setwire(4, wire-distance: 1.3pt), 1, $ U^{2^0} $, $ U^{2^1} $, $ U^{2^2} $,
    1, midstick($ dots $), $ U^{2^{t-1}} $, rstick($|u>$)
)
```



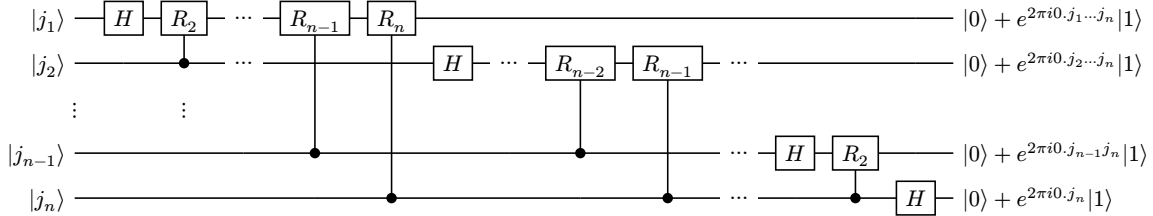
IX.c Quantum Fourier Transform:

Circuit for performing the quantum Fourier transform, reproducing the Figure 5.1 in [1].

```
#quantum-circuit(
  scale: 85%,
  row-spacing: 5pt,
  column-spacing: 8pt,
  lstick($|j_1\rangle$, $H$, $R_2$, midstick($ dots $),
    $R_{(n-1)}$, $R_n$, 8,
    rstick($|0\rangle + e^{(2\pi i \cdot 0 \cdot j_1 \dots j_n)}|1\rangle$, [\ ],
  lstick($|j_2\rangle$, 1, ctrl(-1), midstick($ dots $), 2, $H$, midstick($ dots $),
    $R_{(n-2)}$, $R_{(n-1)}$, midstick($ dots $), 3,
    rstick($|0\rangle + e^{(2\pi i \cdot 0 \cdot j_2 \dots j_n)}|1\rangle$, [\ ],

  setwire(0), midstick($dots.v$), 1, midstick($dots.v$), [\ ],

  lstick($|j_{(n-1)}\rangle$, 3, ctrl(-3), 3, ctrl(-2), 1, midstick($ dots $), $H$,
    $R_2$, 1, rstick($|0\rangle + e^{(2\pi i \cdot 0 \cdot j_{(n-1)} j_n)}|1\rangle$, [\ ],
  lstick($|j_n\rangle$, 4, ctrl(-4), 3, ctrl(-3), midstick($ dots $), 1, ctrl(-1), $H$,
    rstick($|0\rangle + e^{(2\pi i \cdot 0 \cdot j_n)}|1\rangle$)
)
```

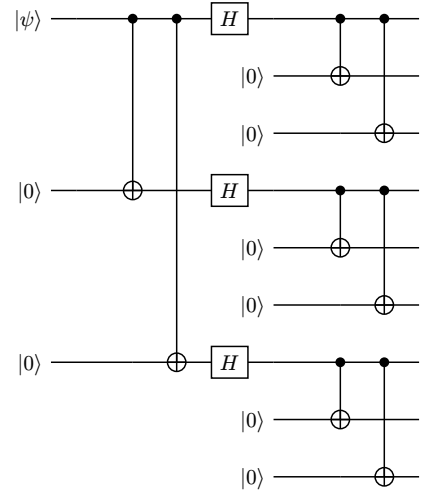


IX.d Shor Nine Qubit Code

Encoding circuit for the Shor nine qubit code. This diagram reproduces Figure 10.4 in [1]

```
#let ancillas = (setwire(0), 5, lstick($|0\rangle$), setwire(1),
  targ(), 2, [\ ],
  setwire(0), 5, lstick($|0\rangle$), setwire(1), 1, targ(), 1)

#quantum-circuit(
  scale-factor: 80%,
  lstick($|\psi\rangle$, 1, 10pt, ctrl(3), ctrl(6), $H$, 1, 15pt,
    ctrl(1), ctrl(2), 1, [\ ],
  ..ancillas, [\ ],
  lstick($|0\rangle$, 1, targ(), 1, $H$, 1, ctrl(1), ctrl(2),
    1, [\ ],
  ..ancillas, [\ ],
  lstick($|0\rangle$, 2, targ(), $H$, 1, ctrl(1), ctrl(2),
    1, [\ ],
  ..ancillas
)
```

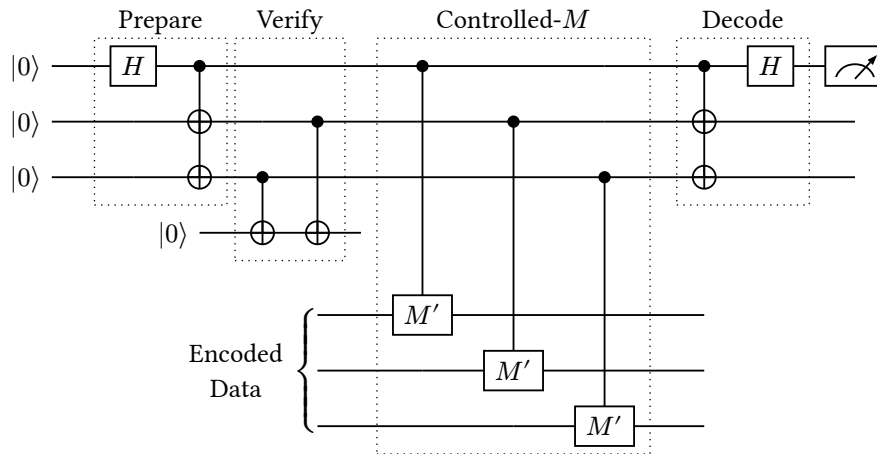


IX.e Fault-Tolerant Measurement

Circuit for performing fault-tolerant measurement (as Figure 10.28 in [1]).

```
#let group = gategroup.with(stroke: (dash: "dotted", thickness: .5pt))

#quantum-circuit(
  row-spacing: 6pt,
  lstick($|0\rangle$, 10pt, group(3, 2, label: (content: "Prepare")), $H$, ctrl(2), 3pt,
  group(4, 2, label: (content: "Verify")), 3,
  group(7, 3, label: (content: [Controlled-$M$])),
  ctrl(4), 2, 10pt, group(3, 2, label: (content: "Decode")), ctrl(2), $H$, meter(), [\ ],
  lstick($|0\rangle$, 1, targ(), 1, ctrl(2), 2, ctrl(4), 1, targ(), 2, [\ ],
  lstick($|0\rangle$, 1, targ(), ctrl(1), 4, ctrl(4), targ(), 2, [\ ],
  setwire(0), 2, lstick($|0\rangle$, setwire(1), targ(), targ(), 1, [\ ], 10pt,
  setwire(0), 4, lstick(aligned=center)[Encoded\ Data], n: 3, setwire(1), 1,
  $M'$, 3, [\ ],
  setwire(0), 5, setwire(1), 2, $M'$, 2, [\ ],
  setwire(0), 5, setwire(1), 3, $M'$, 1,
)
```

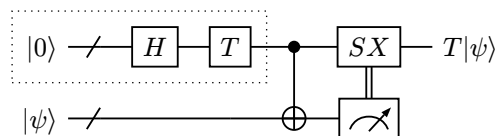


IX.f Fault-Tolerant Gate Construction

The following two circuits reproduce figures from Exercise 10.66 and 10.68 on construction fault-tolerant $\frac{\pi}{8}$ and Toffoli gates in [1].

```
#let group = gategroup.with(stroke: (dash: "dotted", thickness: .5pt))

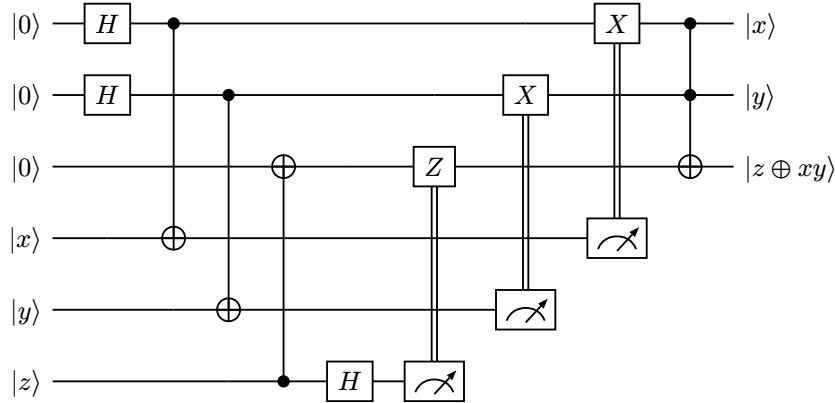
#quantum-circuit(
  group(1, 4, padding: (left: 1.5em)), lstick($|0\rangle$, nwire("")), $H$, $T$,
  ctrl(1), $S\ X$, rstick($T|\psi\rangle$, [\ ],
  lstick($|\psi\rangle$, nwire(""), 2, targ(), meter(target: -1),
)
```




```

#quantum-circuit(
  lstick($|0\rangle$, $H$, ctrl(3), 5, $X$, ctrl(2), rstick($|x\rangle$), [\ ],
  lstick($|0\rangle$, $H$, 1, ctrl(3), 3, $X$, 1, ctrl(0), rstick($|y\rangle$), [\ ],
  lstick($|0\rangle$, 3, targ(), 1, $Z$, 2, targ(), rstick($|z \text{ plus circle } x y\rangle$), [\ ],
  lstick($|x\rangle$, 1, targ(), 5, meter(target: -3), [\ ],
  lstick($|y\rangle$, 2, targ(), 3, meter(target: -3), [\ ],
  lstick($|z\rangle$, 3, ctrl(-3), $H$, meter(target: -3)
)

```

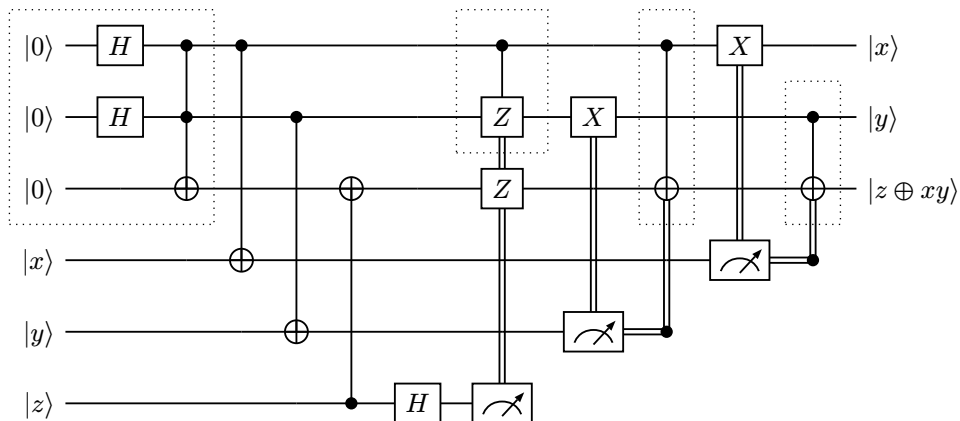


```

#let group = gategroup.with(stroke: (dash: "dotted", thickness: .5pt))

#quantum-circuit(
  group(3, 3, padding: (left: 1.5em)), lstick($|0\rangle$, $H$, ctrl(2), ctrl(3), 3,
    group(2, 1), ctrl(1), 1, group(3, 1), ctrl(2), $X$, 1, rstick($|x\rangle$), [\ ],
    lstick($|0\rangle$, $H$, ctrl(0), 1, ctrl(3), 2, $Z$, $X$, 2, group(2, 1),
      ctrl(1), rstick($|y\rangle$), [\ ],
    lstick($|0\rangle$, 1, targ(), 2, targ(), 1, $Z$, 1, targ(fill: true), 1, targ(fill: true),
      rstick($|z \text{ plus circle } x y\rangle$), [\ ],
    lstick($|x\rangle$, 2, targ(), 6, meter(target: -3), setwire(2), ctrl(-1, wire-count: 2), [\ ],
    lstick($|y\rangle$, 3, targ(), 3, meter(target: -3), setwire(2), ctrl(-2, wire-count: 2), [\ ],
    lstick($|z\rangle$, 4, ctrl(-3), $H$, meter(target: -4)
)

```



BIBLIOGRAPHY

- [1] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*, 2nd ed. Cambridge Cambridge University Press, 2022.