|LIB⟩

Quantum-accelerated scientific computing finally made easy

Matthias Möller

Delft University of Technology
Delft Institute of Applied Mathematics

Joint work with Tim Driebergen, Merel Schalkers, Kelvin Loh, and Richard Versluis

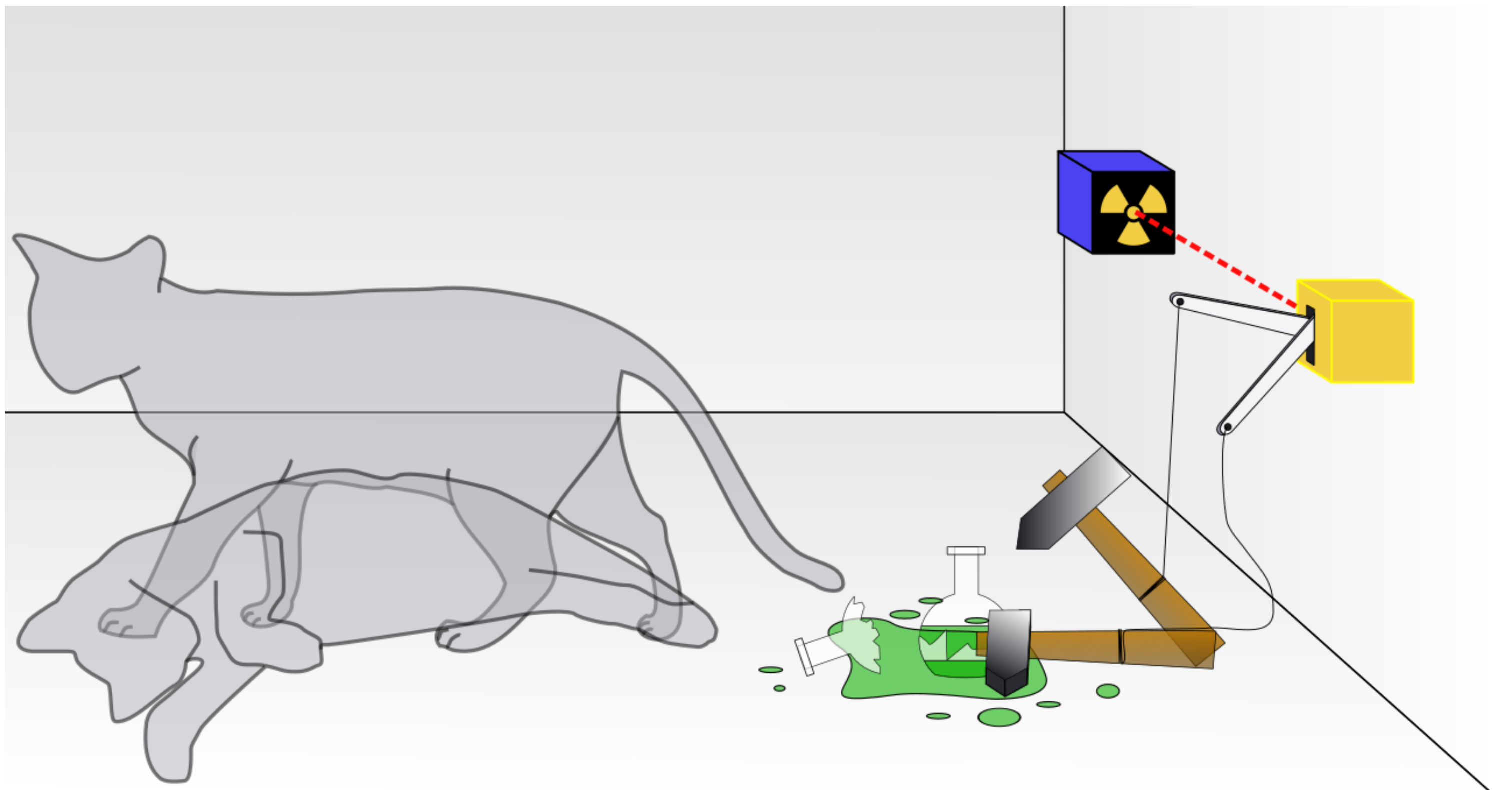
Outlook

- Basic concepts of quantum computing
  - *Single- and multi-qubit states, gates, and simple algorithms*
- Quantum-accelerated scientific computing
  - *NISQ devices, programming models, and potential algorithms*
- LibKet
  - *Design principles and ongoing applications development*
- Conclusion

Basic concepts of quantum computing

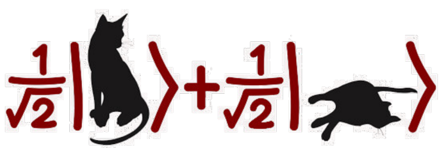
QUANTUM BITS AND GATES

Schrödinger's cat



Schrödinger's cat, cont'd

- Before opening the box:
  superposition *of two states*



- After opening the box:
  collapse *to a single state*

OR

- Further examples of **two-state quantum-mechanical system**
  - spin of an electron (up, down)
  - polarization of a photon (vertical, horizontal)

Quantum bits

- **Qubit**: basic unit of quantum information (quantum version of a bit)

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \qquad \alpha, \beta \in \mathbb{C}, \qquad |\alpha|^2 + |\beta|^2 = 1$$

- Computational **basis**

$$\mathcal{E} = (|0\rangle, |1\rangle) = \left(\begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right)$$

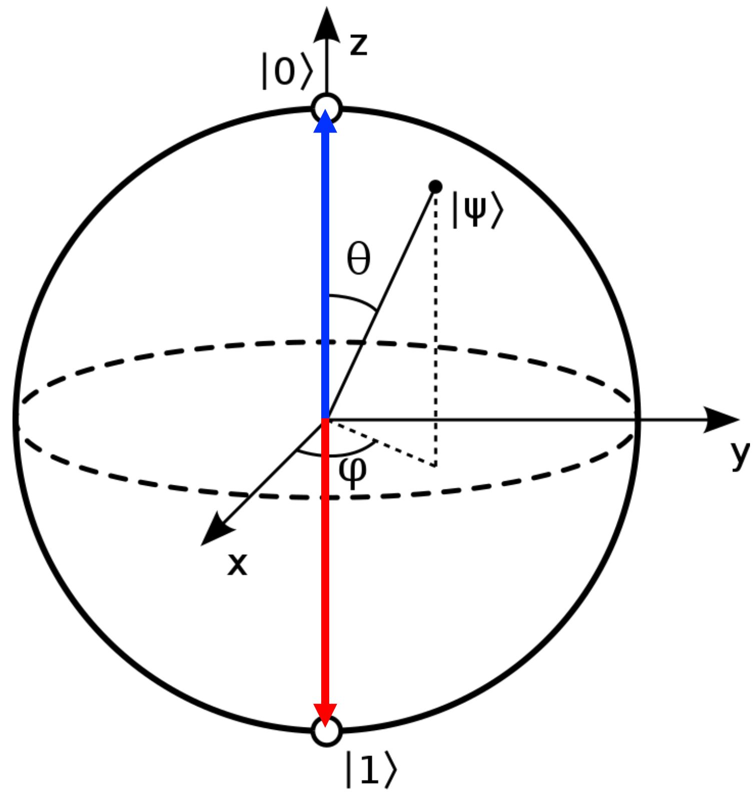
- Coefficients α, β are the **probability amplitues** and $|\alpha|^2$ and $|\beta|^2$ are the **probabilities** of measuring the basis states $|0\rangle$ and $|1\rangle$, respectively

Single-qubit states

- **Bloch sphere**

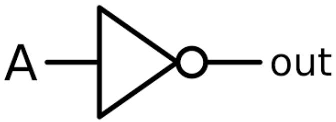
$$|\psi\rangle = e^{i\delta}\left(\cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle\right)$$

  - polar angle $\theta \in [0, \pi]$
  - azimutal angle $\varphi \in [0,2\pi)$
  - ~~global phase δ~~



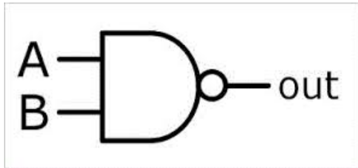
Classical gates

- **NOT**



A	out
0	1
1	0

- **NAND**

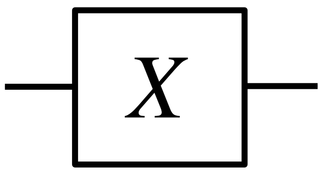


A	B	out
0	0	1
0	1	1
1	0	1
1	1	0

- Logical operations based on truth tables
- Most classical gates are not reversible

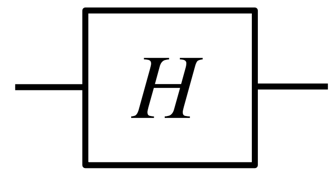
Quantum gates

- **Pauli X**



$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

- **Hadamard**

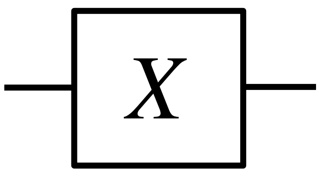


$$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

- Unitary operations represented by unitary matrices
- All quantum gates are reversible, e.g. $HH^{\dagger} = I$
- Universal gate set $\{H, S, T, CNOT\}$

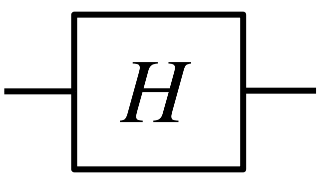
Single-qubit gates

$|0\rangle$

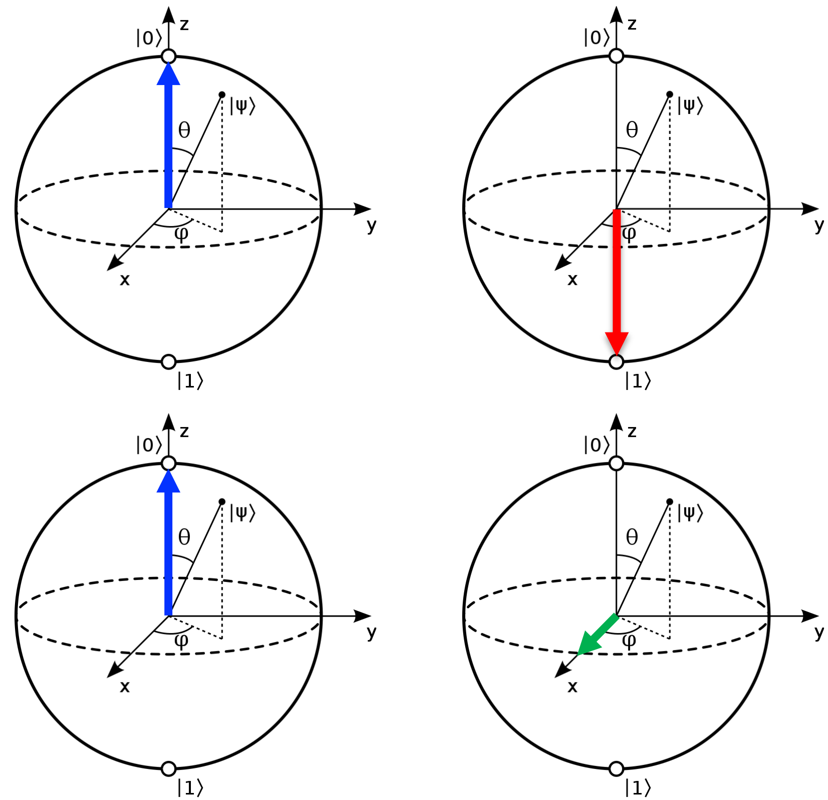


$|1\rangle$

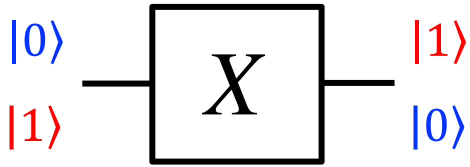
$|0\rangle$

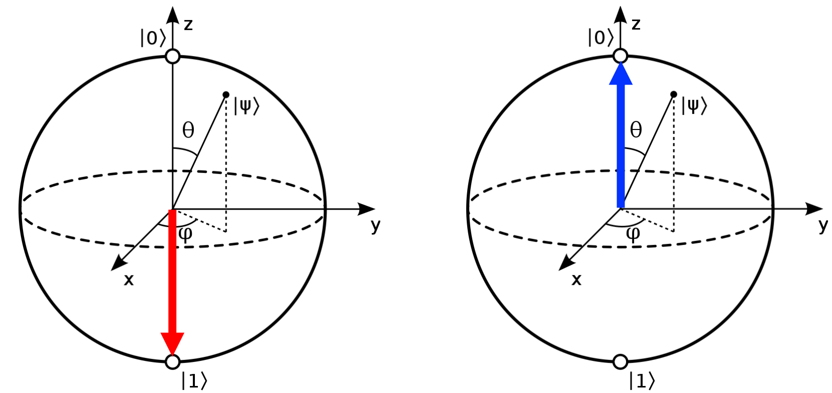


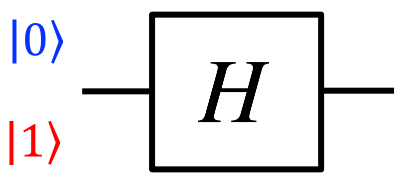
$$|+\rangle := \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$



Single-qubit gates

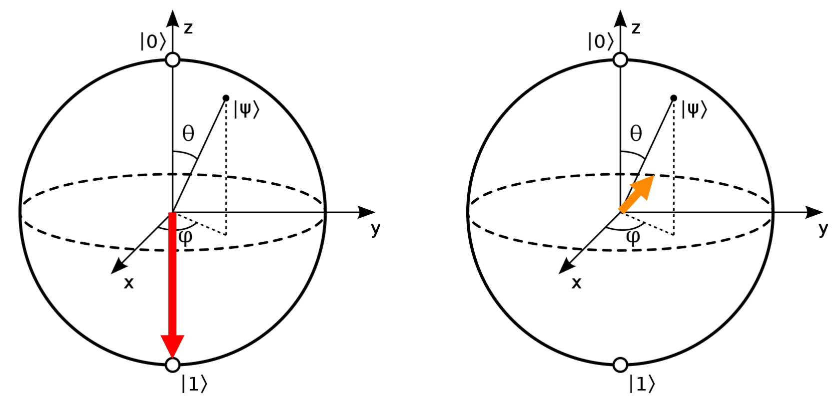




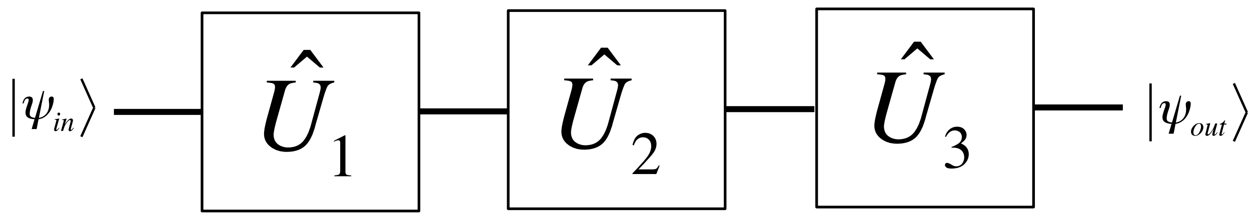


$$|+\rangle := \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

$$|-\rangle := \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$



Single-qubit circuits



- Single-qubit gates $\hat{U}_k$ are **unitary matrices**, i.e.

$$\hat{U}_k \hat{U}_k^\dagger = \hat{U}_k^\dagger \hat{U}_k = \hat{I}$$

- Quantum circuits are sequences of matrix-vector multiplications

$$|\psi_{out}\rangle = \hat{U}_3 \hat{U}_2 \hat{U}_1 \; |\psi_{in}\rangle$$

Multi-qubit states

- $|\psi_0\rangle = \alpha_0|0\rangle + \beta_0|1\rangle = \alpha_0\begin{pmatrix}1\\0\end{pmatrix} + \beta_0\begin{pmatrix}0\\1\end{pmatrix}$
- $|\psi_1\rangle = \alpha_1|0\rangle + \beta_1|1\rangle = \alpha_1\begin{pmatrix}1\\0\end{pmatrix} + \beta_1\begin{pmatrix}0\\1\end{pmatrix}$

Tensor product

$$|A\rangle \otimes |B\rangle = \begin{bmatrix} a_{11}B & a_{12}B \\ a_{21}B & a_{22}B \end{bmatrix}$$

- **Tensor product** of two single-qubit states

$$|\psi_0\rangle \otimes |\psi_1\rangle = \alpha_0\alpha_1|00\rangle + \alpha_0\beta_1|01\rangle + \beta_0\alpha_1|10\rangle + \beta_0\beta_1|11\rangle =: |\psi_0\psi_1\rangle$$

with

$$|\alpha_0\alpha_1|^2 + |\alpha_0\beta_1|^2 + |\beta_0\alpha_1|^2 + |\beta_0\beta_1|^2 =$$
$$|\alpha_0|^2(|\alpha_1|^2 + |\beta_1|^2) + |\alpha_1|^2(|\alpha_1|^2 + |\beta_1|^2) = 1$$

Multi-qubit states, cont'd

- **Tensor product** of n single-qubit states

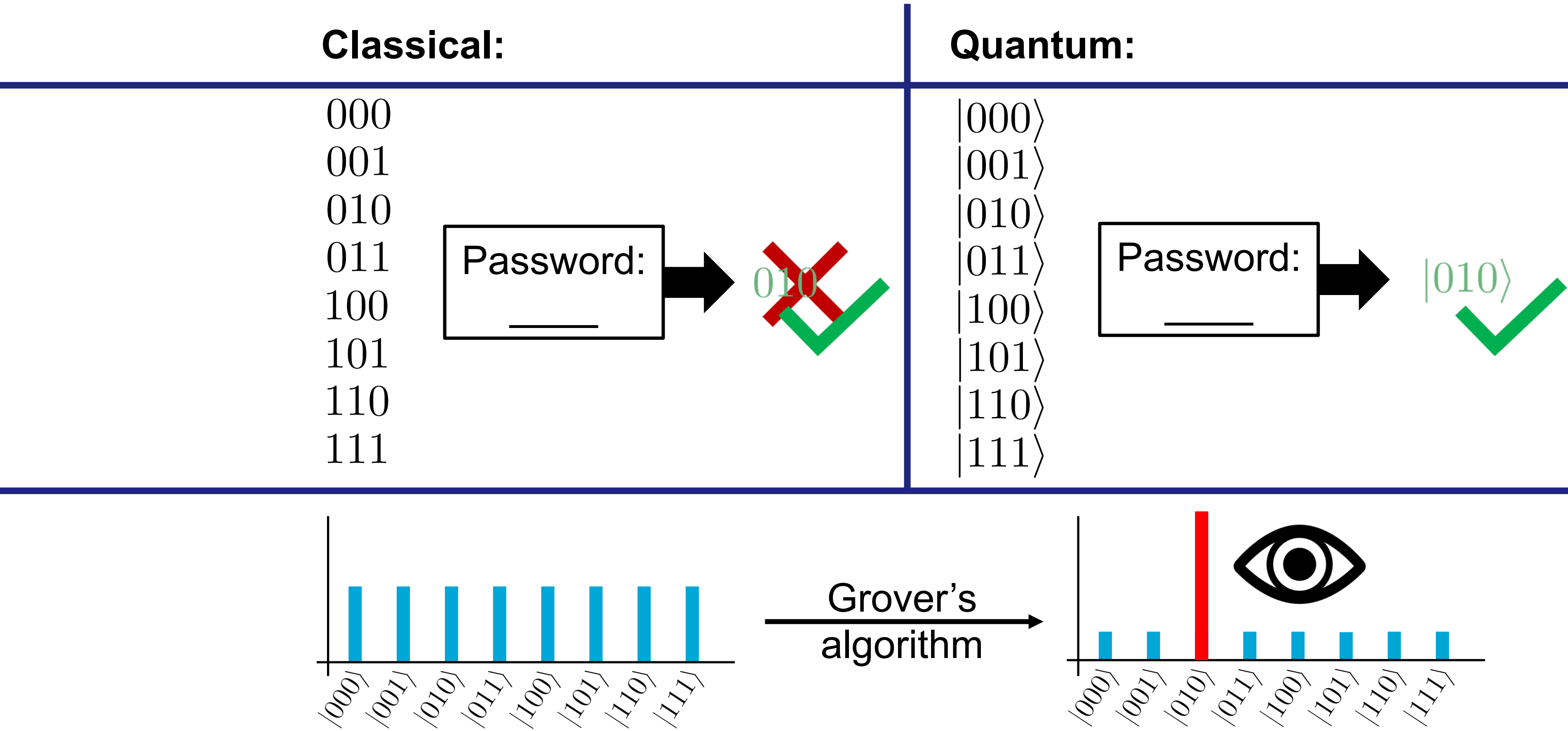
$$|\psi_0 \dots \psi_n\rangle = \gamma_{0\dots00}|0 \dots 00\rangle + \gamma_{0\dots01}|0 \dots 01\rangle + \cdots + \gamma_{1\dots11}|1 \dots 11\rangle$$

- An n-qubit register can hold the 2^n inputs 'simultaneously' in superposition

- **A word of caution**: it is impossible to obtain the γ's; one obtains a single binary answer, say, $|001101\rangle$ with probability $|\gamma_{001101}|^2$ upon measuring

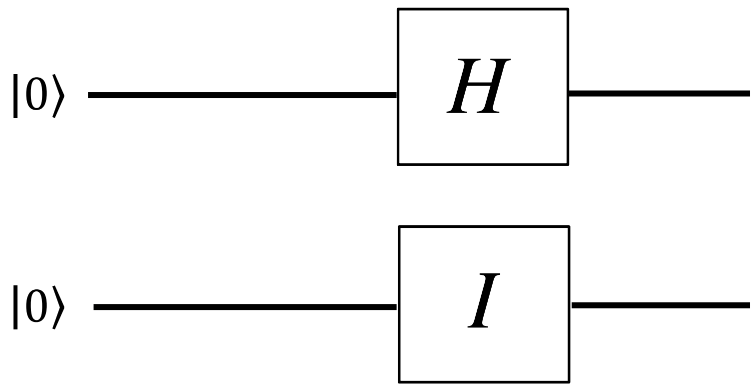
- A single run of a quantum circuit is not very useful; many runs are required to measure the correct answer to the problem with sufficient certainty

Example: 3-bit password



Multi-qubit gates

$|\Psi_{in}\rangle$



$|\Psi_{out}\rangle = H \otimes I|\Psi_{in}\rangle$

$$H \otimes I|00\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \frac{|00\rangle + |10\rangle}{\sqrt{2}} = \frac{\big(|0\rangle + |1\rangle\big) \otimes |0\rangle}{\sqrt{2}}$$

Basic concepts of quantum computing

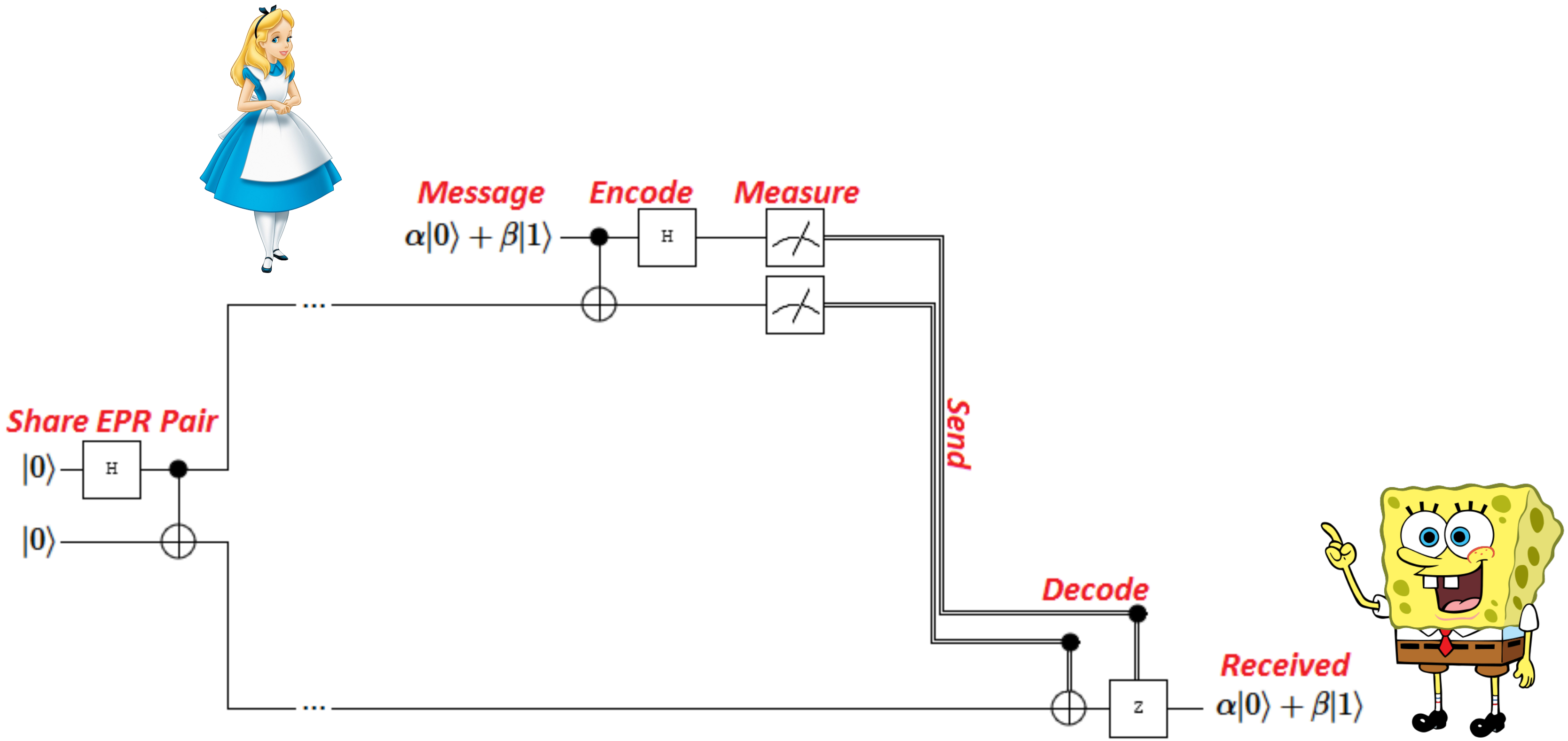
SIMPLE QUANTUM ALGORITHMS

Bell state

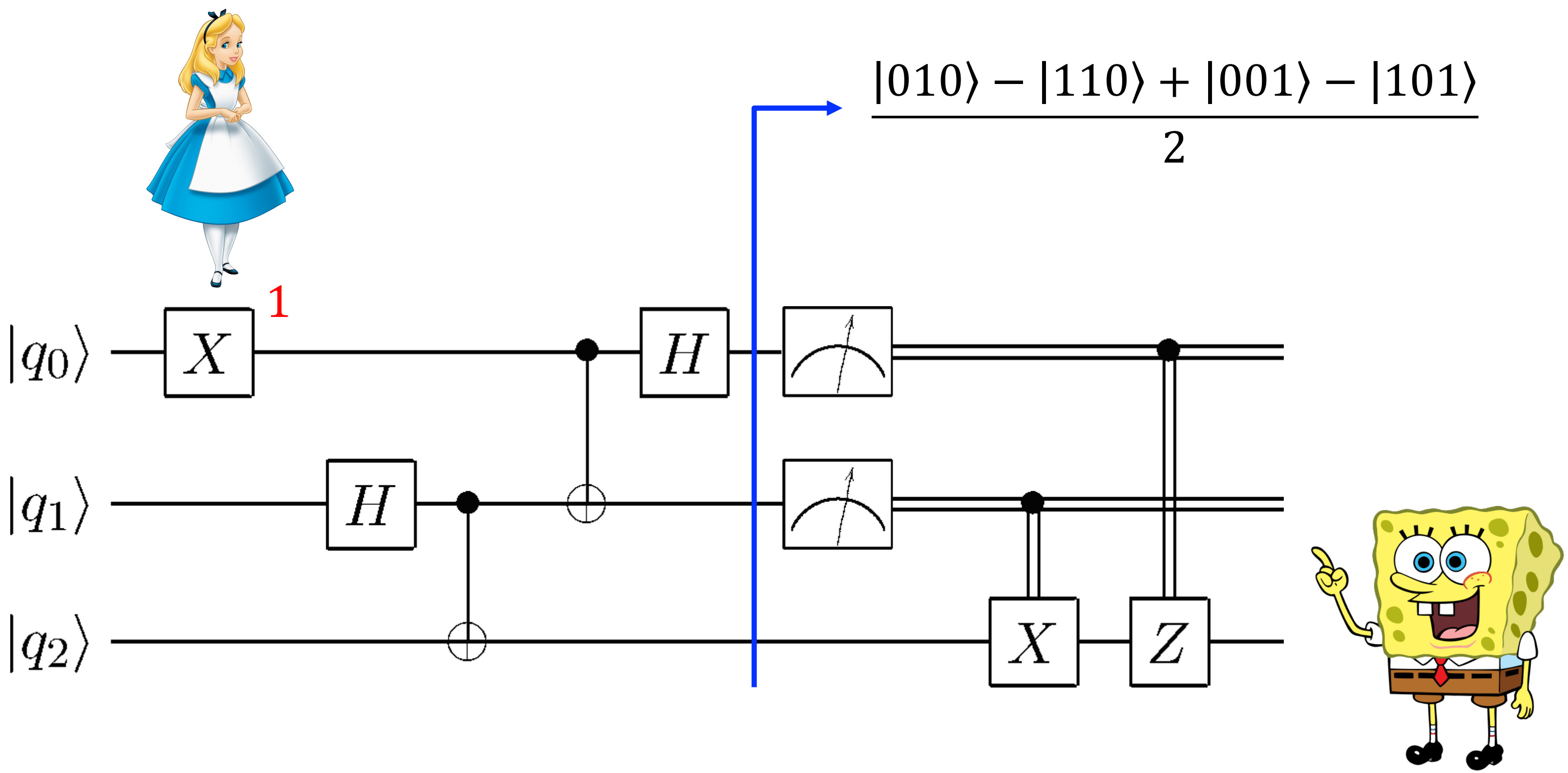
$$CNOT(H \otimes I)|00\rangle = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

- The Bell state is maximally entangled. By measuring one of the two qubits one knows the value of the other qubit without a further measurement

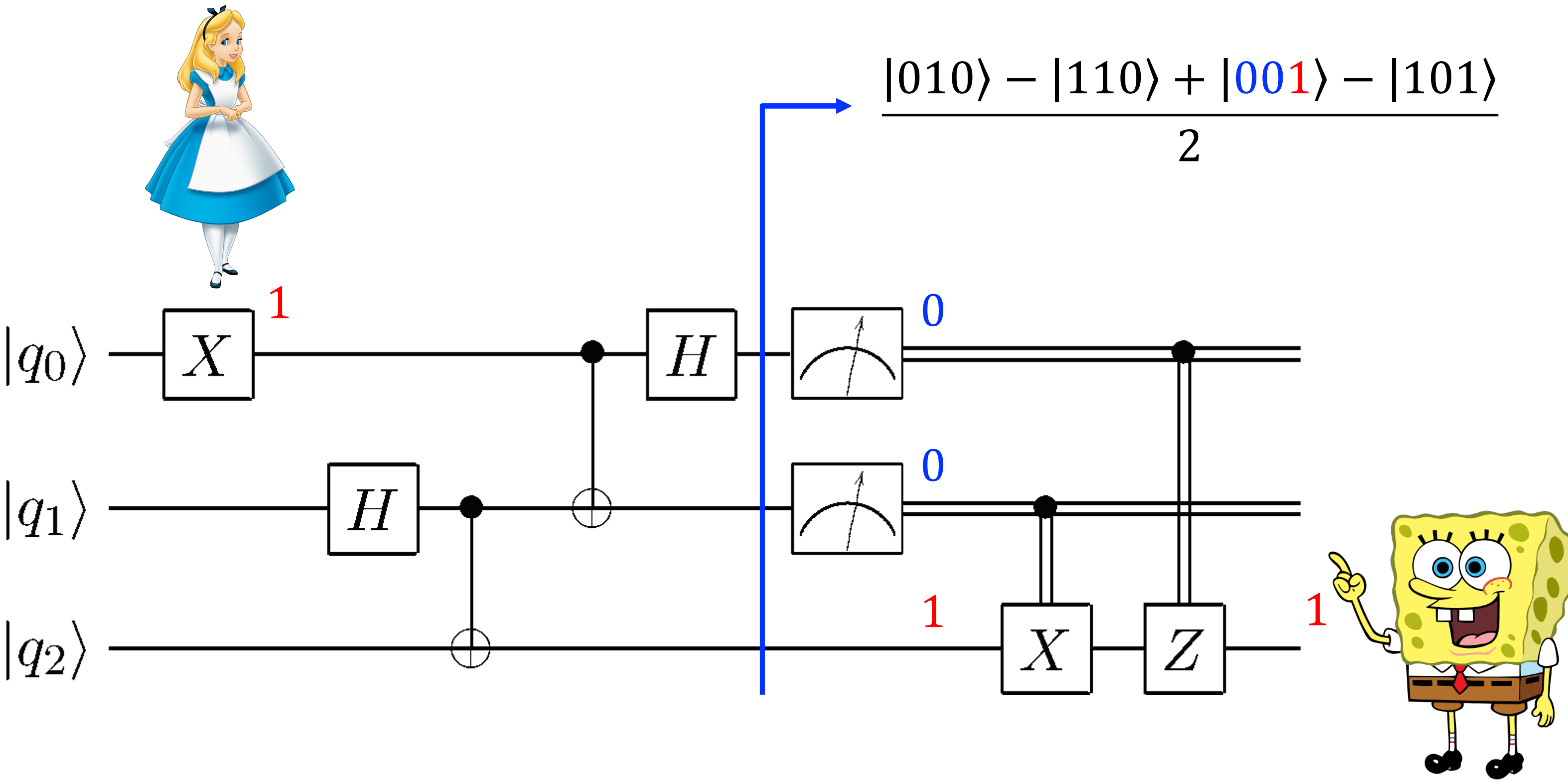
Quantum teleportation



Quantum teleportation

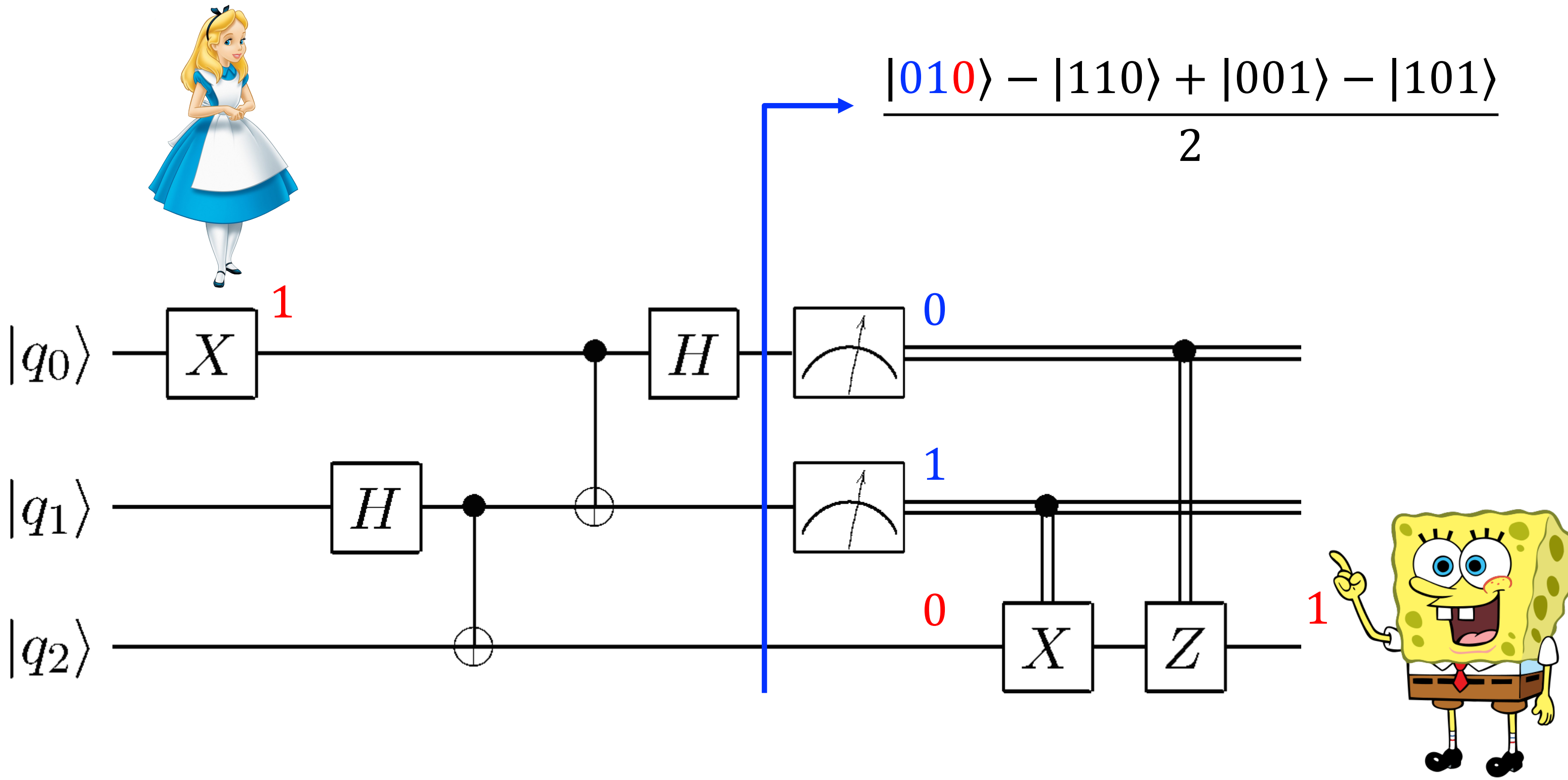


Quantum teleportation

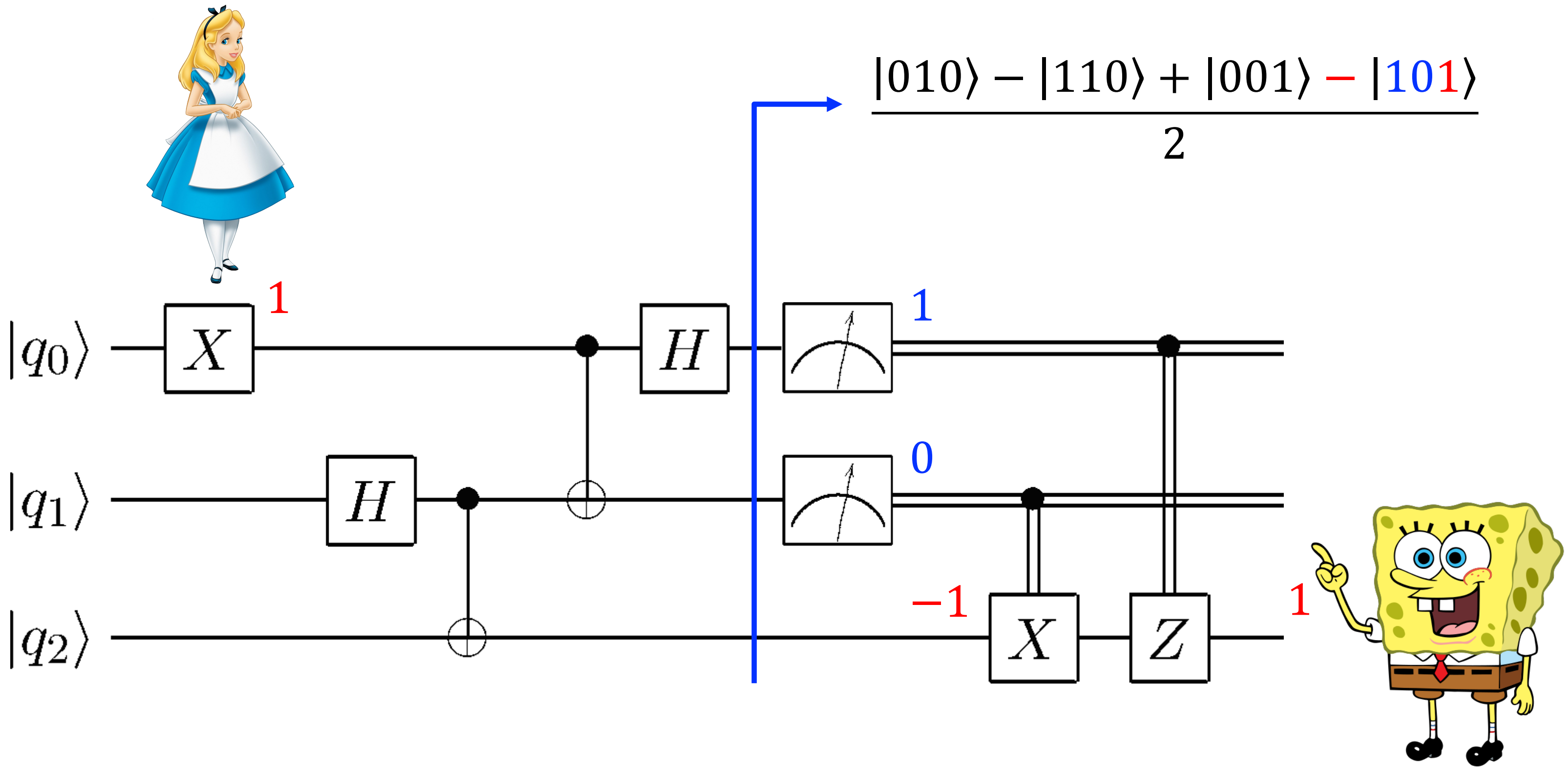


$$\frac{|010\rangle - |110\rangle + |001\rangle - |101\rangle}{2}$$

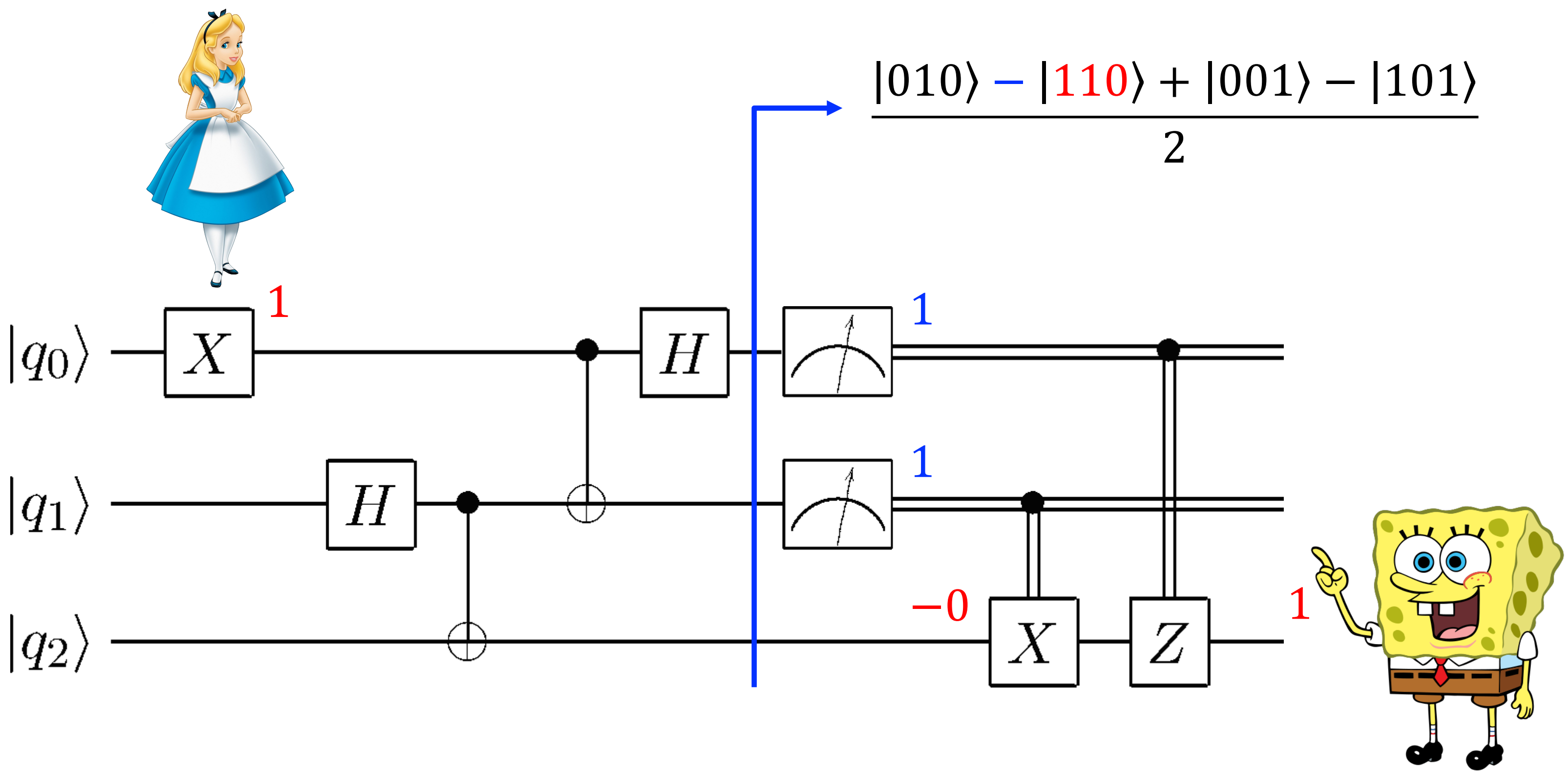
Quantum teleportation



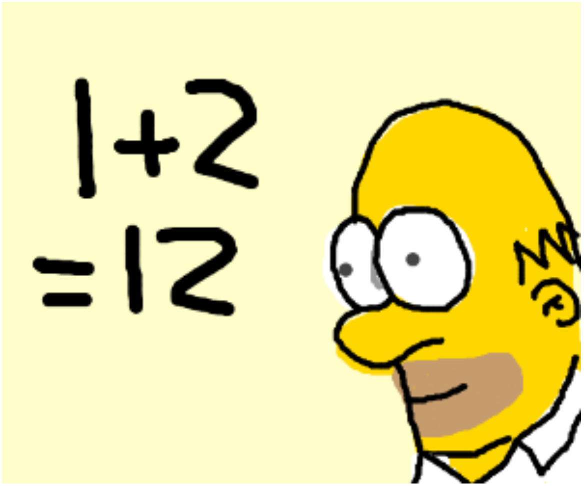
Quantum teleportation



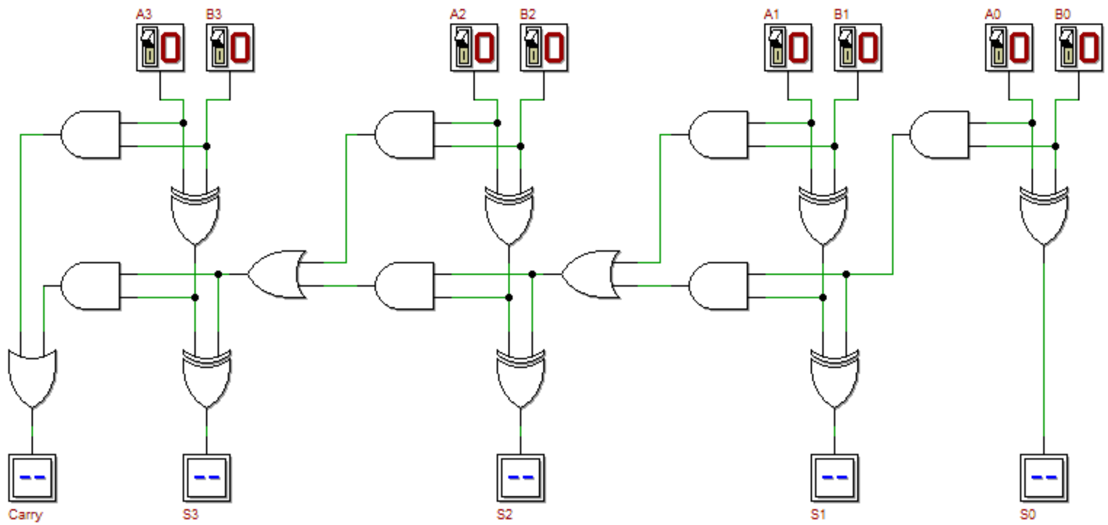
Quantum teleportation



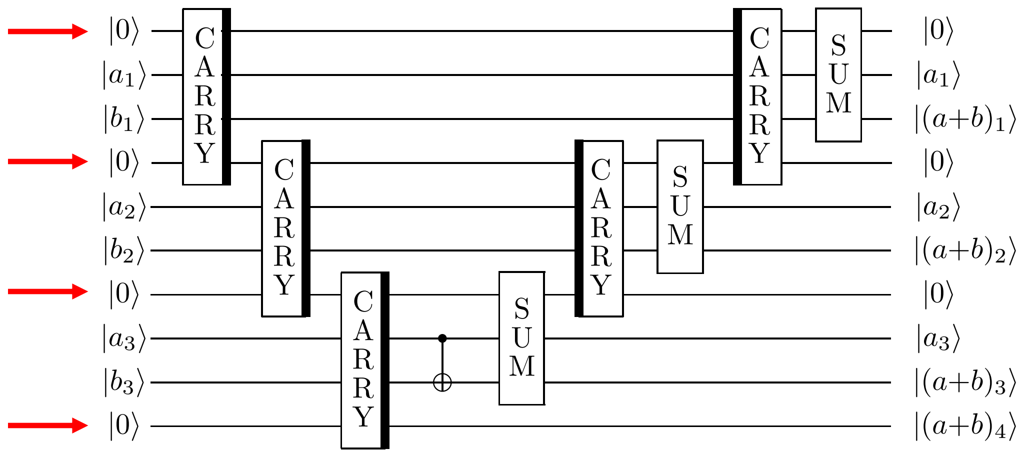
How difficult can it be to add two integers?

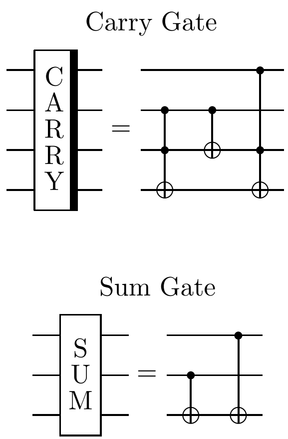


Classical integer adder



A first quantum integer adder

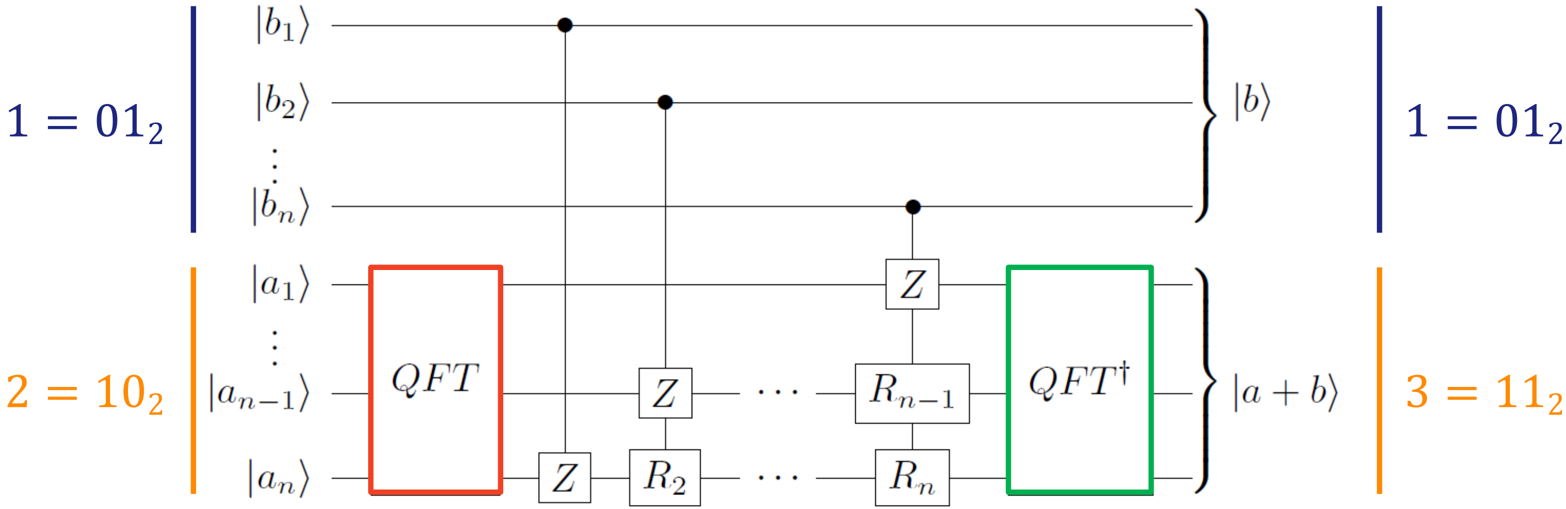


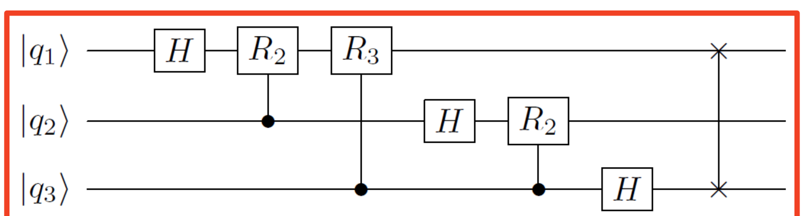


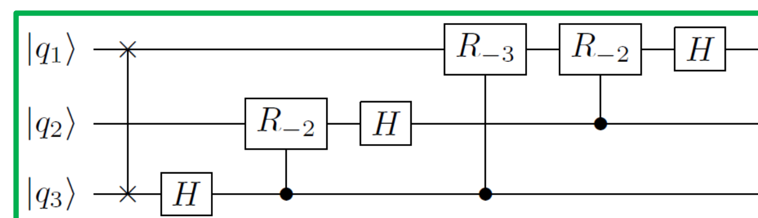
n extra ancilla qubits needed ☹

Cuccaro et al.: A new quantum ripple-carry addition circuit, arXiv:quant-ph/0410184, 2004

Another quantum integer adder







Draper: Addition on a quantum computer, arXiv:quant-ph/0008033, 2000

Towards a practical quantum integer adder

1000 QX simulator runs with depolarizing noise error model

QInt<n>	0,1		$10^{-\frac{3}{2}}$		0,01		$10^{-\frac{5}{2}}$	
1	0.27045	0.3793	0.50545	0.2752	0.78965	0.1233	0.92285	0.0463
2	0.134061	0.221523	0.165182	0.209176	0.451353	0.134284	0.762621	0.0570876
3	0.0601436	0.112097	0.0683512	0.116162	0.191802	0.105916	0.540766	0.0754021
4	0.0336509	0.0611537	0.0351125	0.0589036	0.064375	0.0645881	0.306778	0.0802711
5					0.0224336	0.031892	0.154869	0.0575671
6					0.00798384	0.0176539	0.0654961	0.033179
7					0.00398747	0.0076473	0.0252142	0.0167067
8					0.00254026	0.00363275	0.00834128	0.00823629

Standard circuit: prob. correct (left), largest prob. wrong answer (right)

M. Looman: Implementation and Analysis of an Algorithm on Positive Integer Addition for Quantum Computing, Bachelor thesis, 2018

Towards a practical quantum integer adder

1000 QX simulator runs with depolarizing noise error model

QInt<n>	0,1		$10^{-\frac{3}{2}}$		0,01		$10^{-\frac{5}{2}}$	
1	0.29475	0.3695	0.54555	0.27185	0.8158	0.11735	0.93645	0.04195
2	0.110416	0.230068	0.239152	0.203304	0.569495	0.115691	0.837026	0.0445888
3	0.0581316	0.114572	0.096711	0.122477	0.341537	0.102147	0.697436	0.0509187
4	0.0259028	0.0583002	0.0382769	0.0672328	0.183066	0.0726129	0.543162	0.0579935
5					0.0839273	0.0450361	0.407117	0.0574072
6					0.0412412	0.0270095	0.283642	0.049151
7					0.0177059	0.0131818	0.191996	0.0404665
8					0.00647699	0.00675828	0.116269	0.0290022

Optimized circuit: prob. correct (left), largest prob. wrong answer (right)

M. Looman: Implementation and Analysis of an Algorithm on Positive Integer Addition for Quantum Computing, Bachelor thesis, 2018

Quantum-accelerated scientific computing

NISQ DEVICES, PROGRAMMING MODELS, AND ALGORITHMS

NISQ era

John Preskill

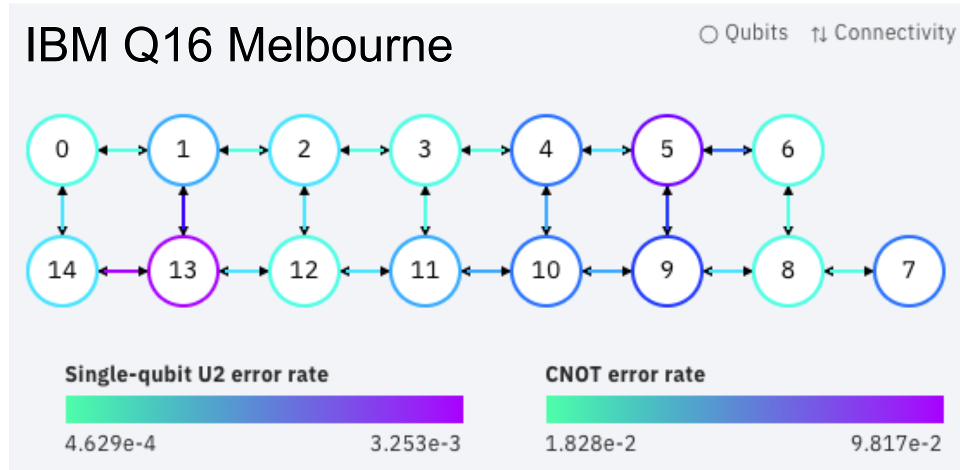
- **Noisy Intermediate-Scale Quantum technology**
  arXiv:1801.00862, 2018

  - **Noisy** emphasizes that we'll have imperfect control over qubits
    - application of $R_\phi = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix}$ is inaccurate, i.e. $R_{\phi \pm \epsilon}$
    - quantum state decoheres, i.e. $|\alpha|^2 + |\beta|^2 \neq 1$

  - **Intermediate-Scale** refers to the size of the current and near-future quantum computers which will have between 50 to a few hundred qubits

Quantum processors

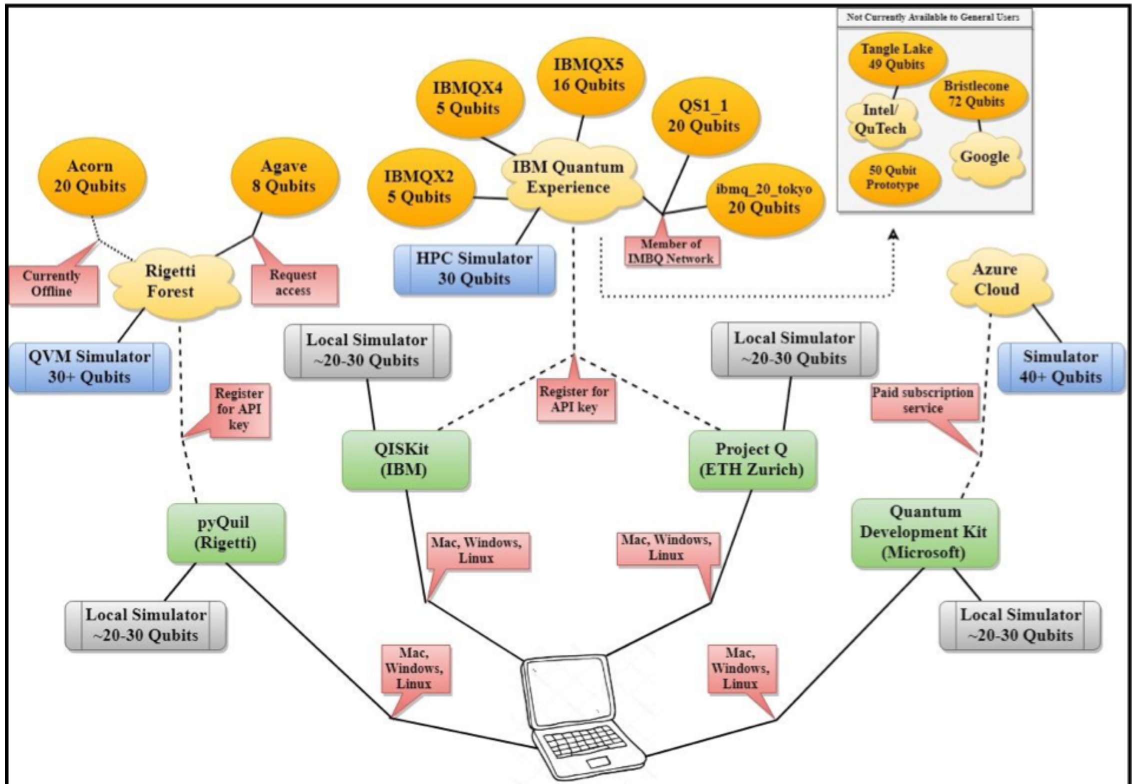
Manufacturer	#qubits
IBM	5-53
Rigetti	8-32
Intel	17-49
Google	20-72



Rigetti's Aspen-7-28Q-A

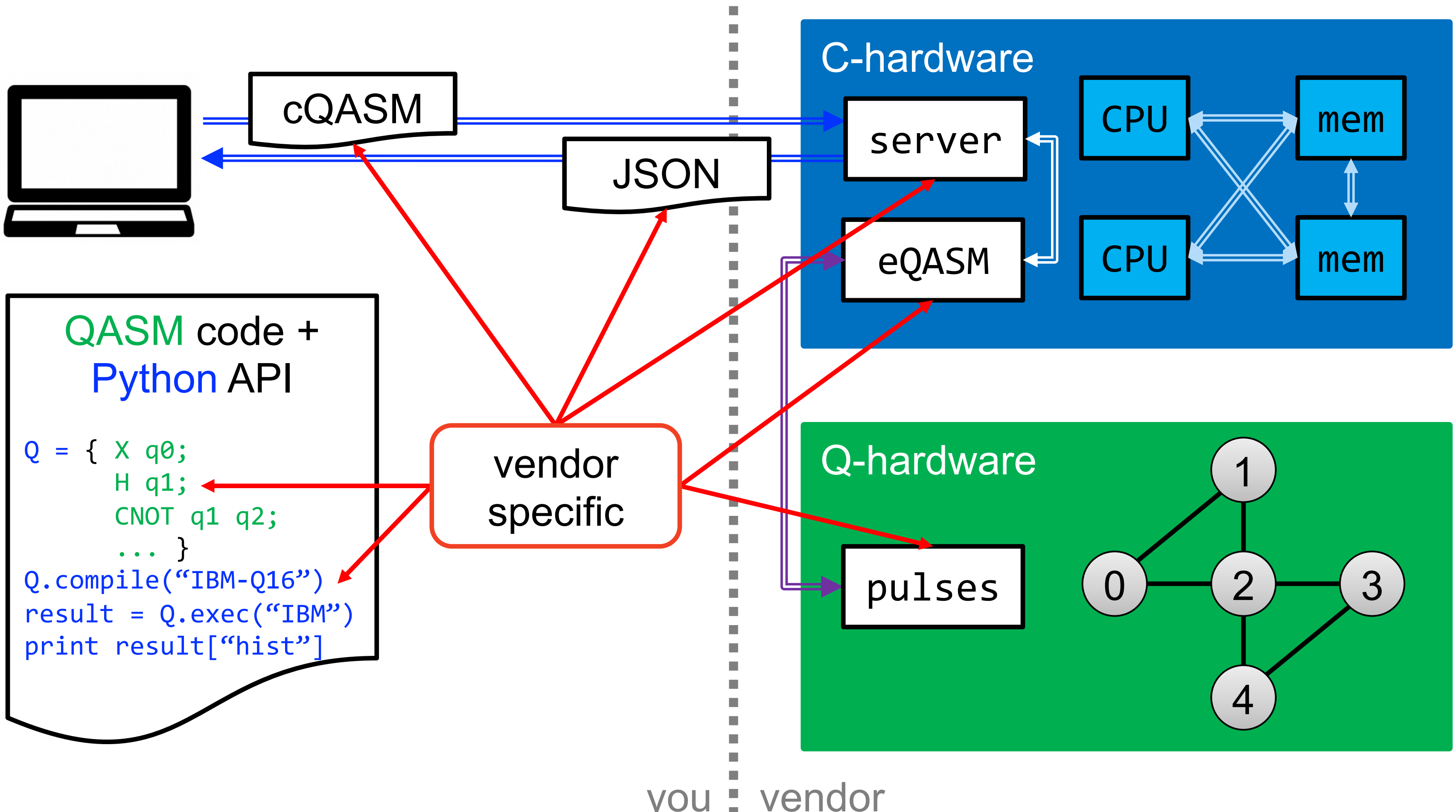
- **In-memory computing**
- Optimal placement and routing of information is crucial; many extra ops

Quantum software platforms

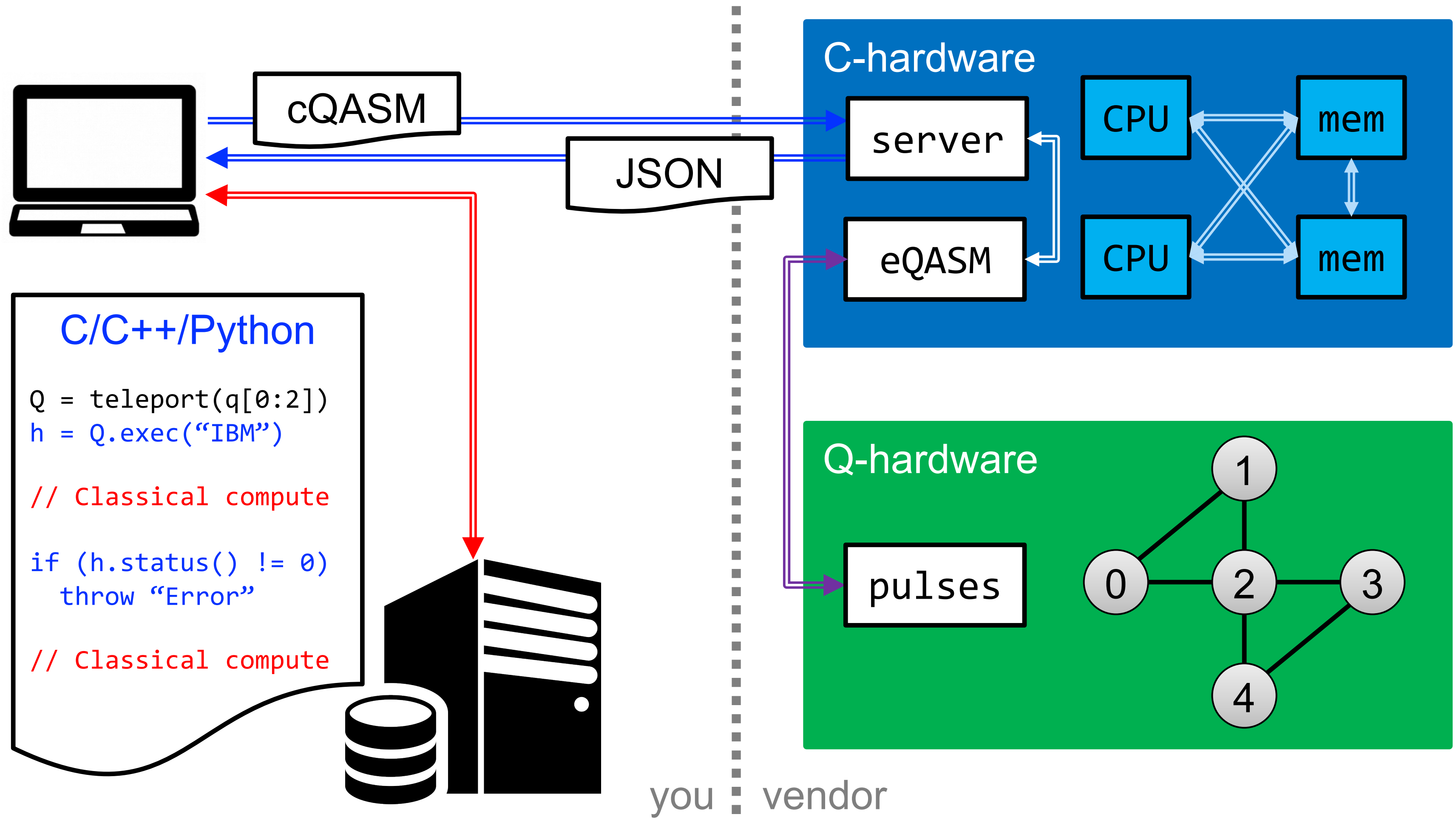


LaRose: Overview and Comparison of Gate Level Quantum Software Platforms, arXiv:1807.02500, 2018

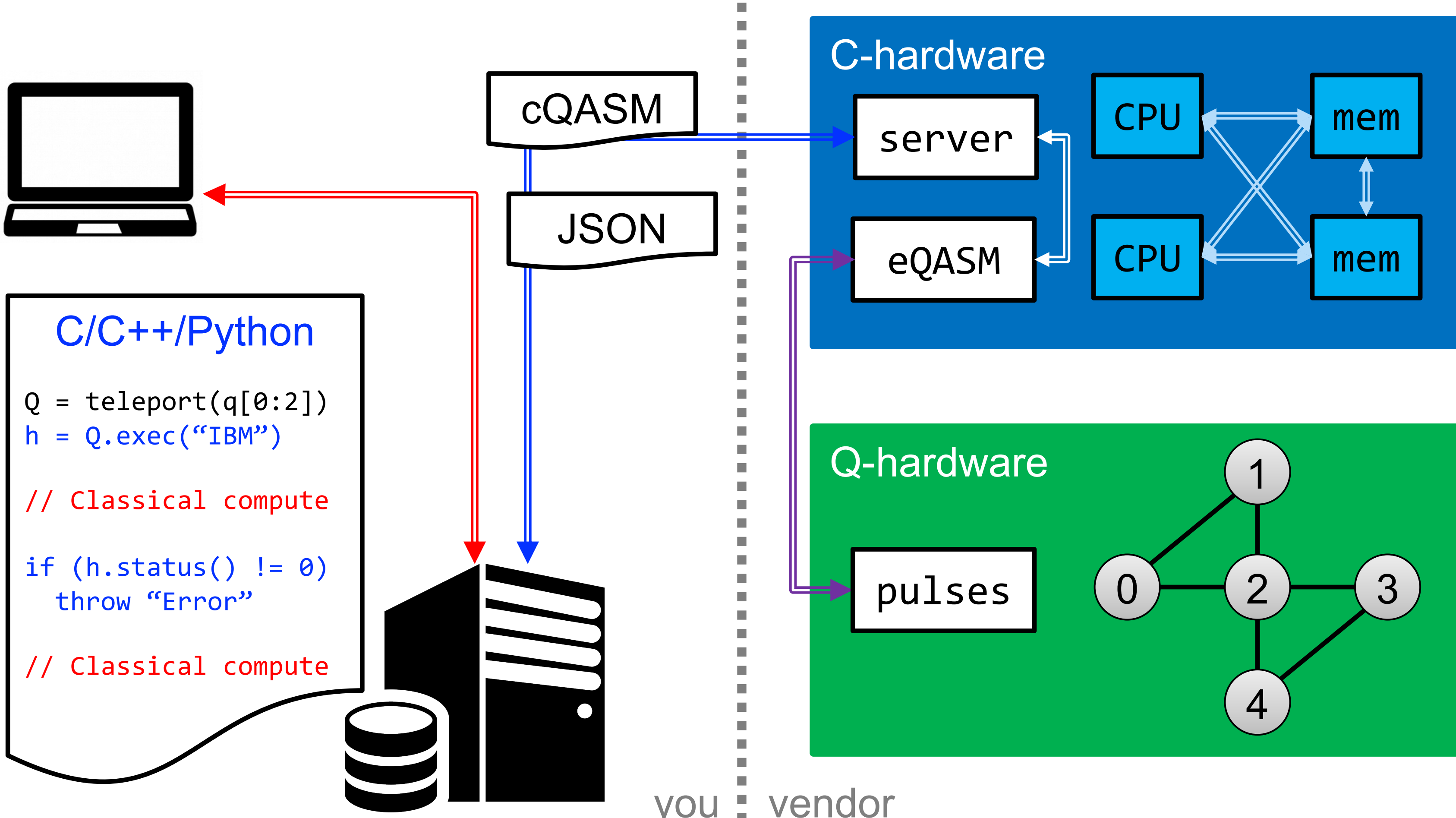
Q-programming model – today



Q-accelerated programming model – our vision



Q-accelerated programming model – our vision



Quantum algorithms with potential use in SciComp

- **Quantum linear solvers**

  - HHL-type 'solver' algorithms: $x^\dagger M x$ such that $Ax = b$
    - sparse matrices [Harrow, Hassidim, Lloyd 2009] $O(\log(N)\kappa^2/\epsilon)$
    - dense matrices [Wossnig et al. 2018] $O(\sqrt{N}\log(N)\kappa^2/\epsilon)$

  - Hybrid Variational QC Algorithms (HVQCA)
    - sparse matrices [Bravo-Prieto et al. 2019 & Xu et al. 2019]
      linear scaling in κ and super-linear scaling in #qubits

S. Aaronson: Read the fine print. Nature Physics 11, 2015.

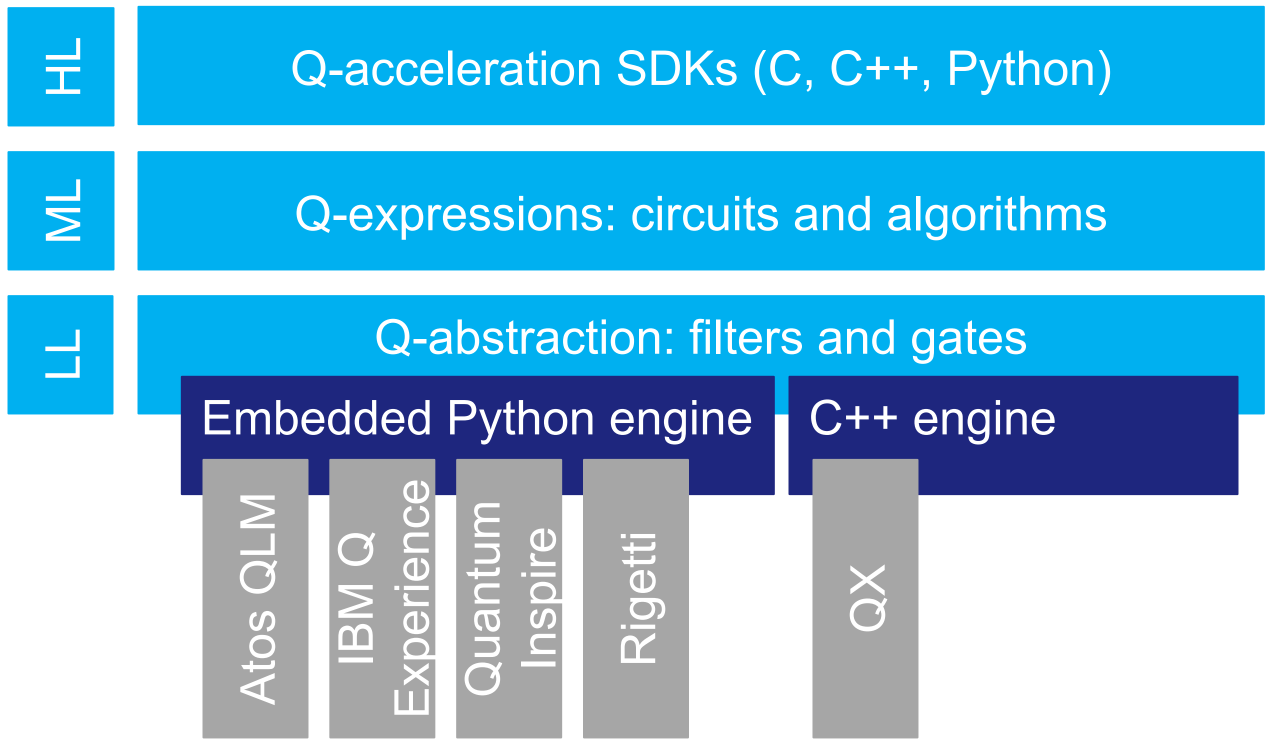
Quantum algorithms with potential use in SciComp, cont'd

- **Quantum algorithms for …**
  - linear differential equations [Berry 2010, Xin et al. 2018]
  - nonlinear differential equations [Leyton, Osborne 2008]
  - Poisson equation [Cao et al. 2013]
  - principal component analysis [Lloyd et al. 2014]
  - data fitting [Wiebe et al. 2012]
  - machine learning [Lloyd et al. 2013, Adcock et al. 2015, Biamonte et al. 2017, Schuld et al. 2018, Perdomo-Ortiz et al. 2018, …]

LibKet: The Kwantum expression template LIBrary

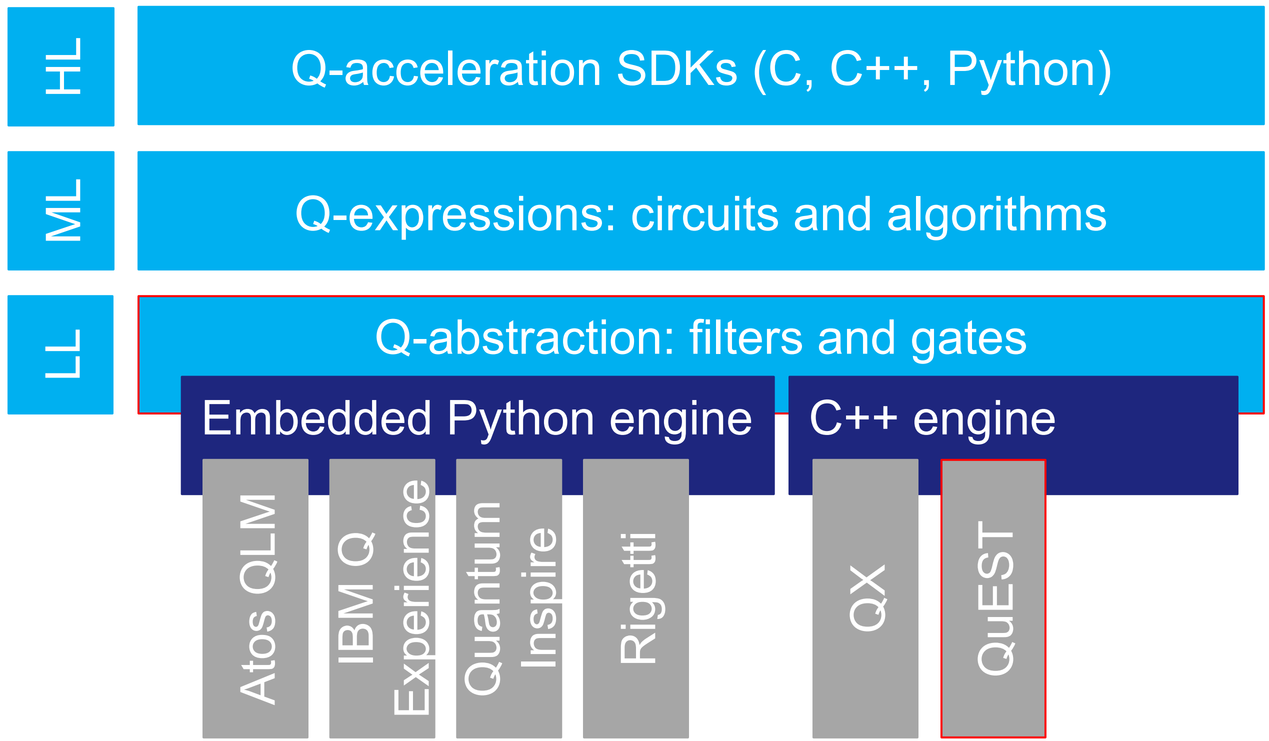
DESIGN PRINCIPLES

Kwantum expression template LIBrary



MM, Schalkers: LibKet: A cross-platform programming framework for quantum-accelerated scientific computing, submitted to ICCS 2020

Kwantum expression template LIBrary



MM, Schalkers: LibKet: A cross-platform programming framework for quantum-accelerated scientific computing, submitted to ICCS 2020

Expression templates

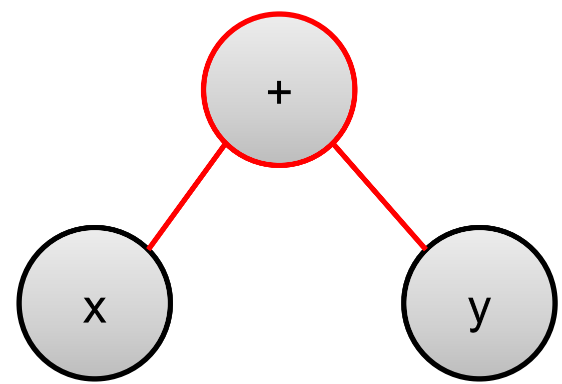
- C++ metaprogramming technique to create lightweight expressions whose evaluation is delayed until their values are really needed

```
Vector x(n), y(n);
```

Expression templates

- C++ metaprogramming technique to create lightweight expressions whose evaluation is delayed until their values are really needed

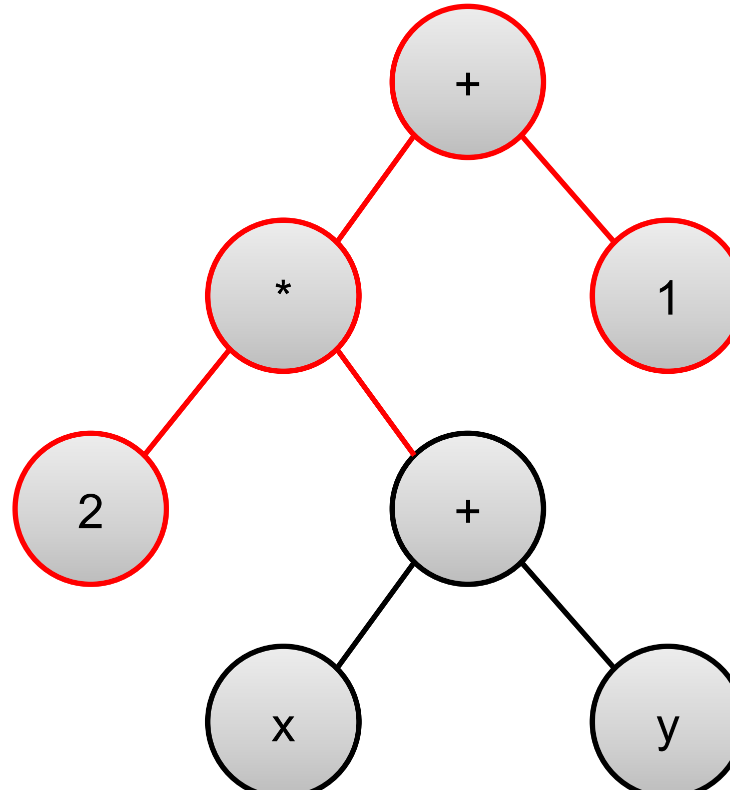
```
Vector x(n), y(n);
auto e0 = x + y;
```



Expression templates

- C++ metaprogramming technique to create lightweight expressions whose evaluation is delayed until their values are really needed

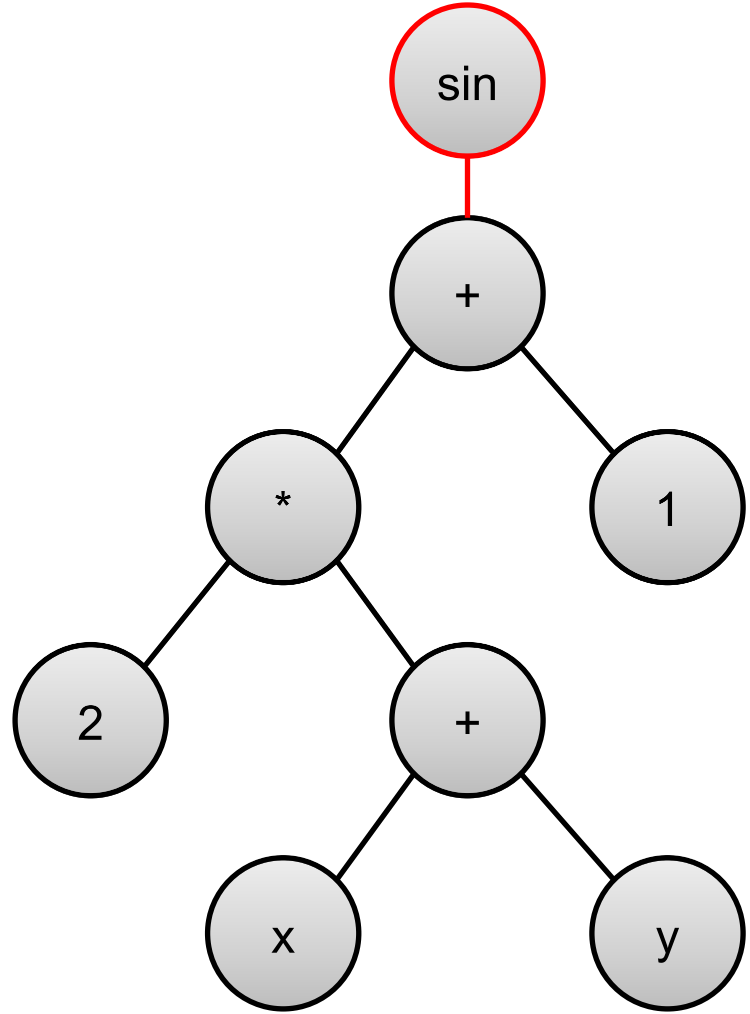
```
Vector x(n), y(n);
auto e0 = x + y;
auto e1 = 2*e0 + 1;
```



Expression templates

- C++ metaprogramming technique to create lightweight expressions whose evaluation is delayed until their values are really needed

```
Vector x(n), y(n);
auto e0 = x + y;
auto e1 = 2*e0 + 1;
auto e2 = sin(e1);
```

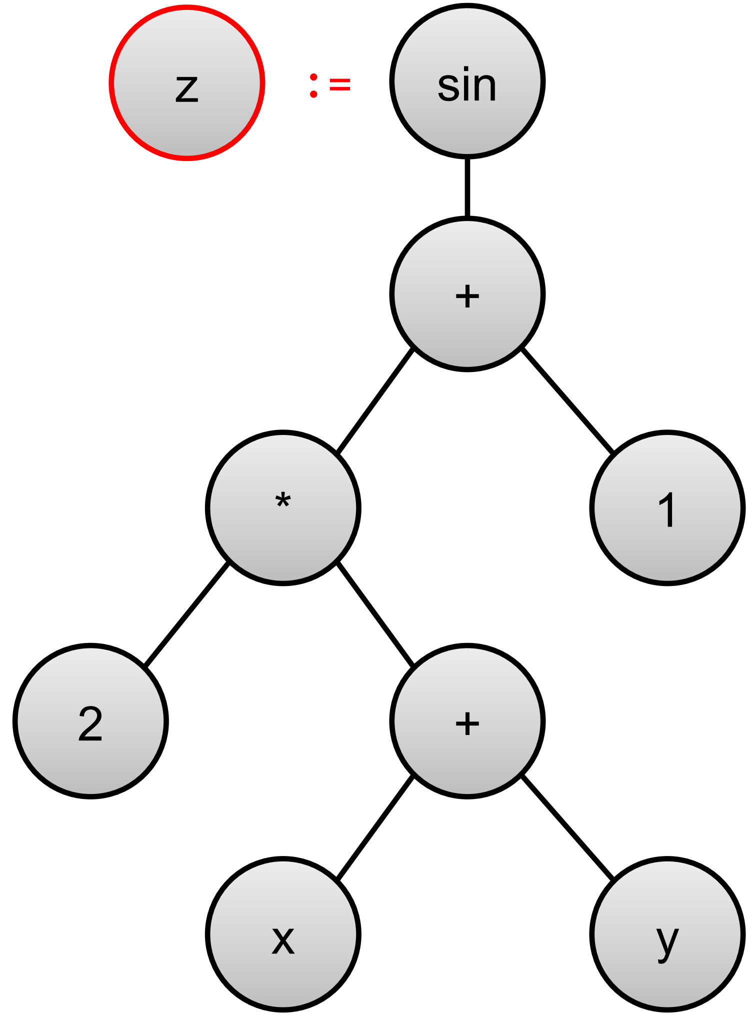


Expression templates

- C++ metaprogramming technique to create lightweight expressions whose evaluation is delayed until their values are really needed

```
Vector x(n), y(n);
auto e0 = x + y;
auto e1 = 2*e0 + 1;
auto e2 = sin(e1);
Vector z = e2;
```

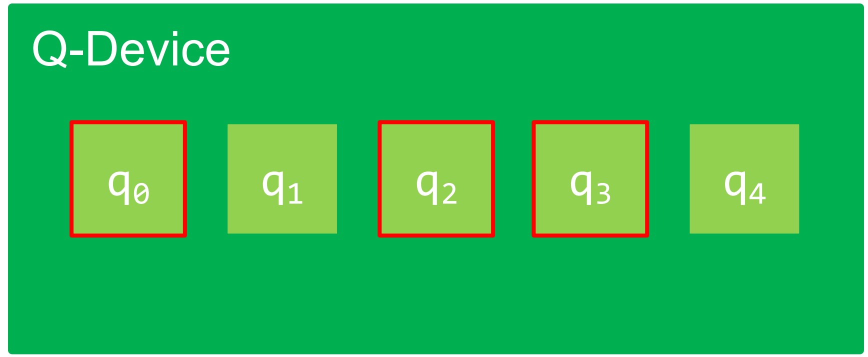
```
-> z[i] = sin(2*(x[i]+y[i])+1);
```



Filters – views on the global Q-memory

- Starting from the full Q-memory filters restrict qubits step by step

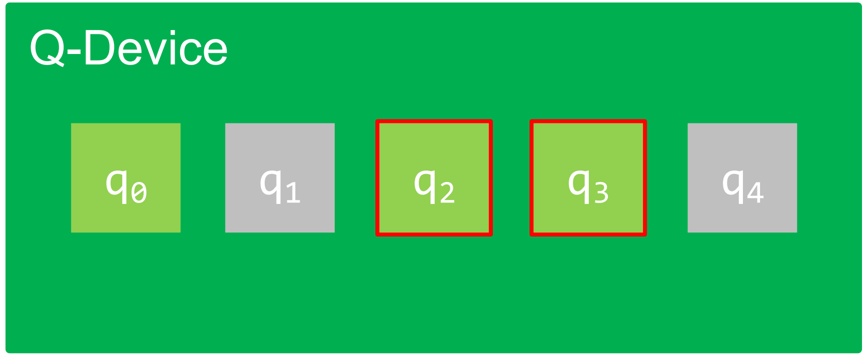
```
auto f0 = select<0,2,3>();
```



Filters – views on the global Q-memory

- Starting from the full Q-memory filters restrict qubits step by step

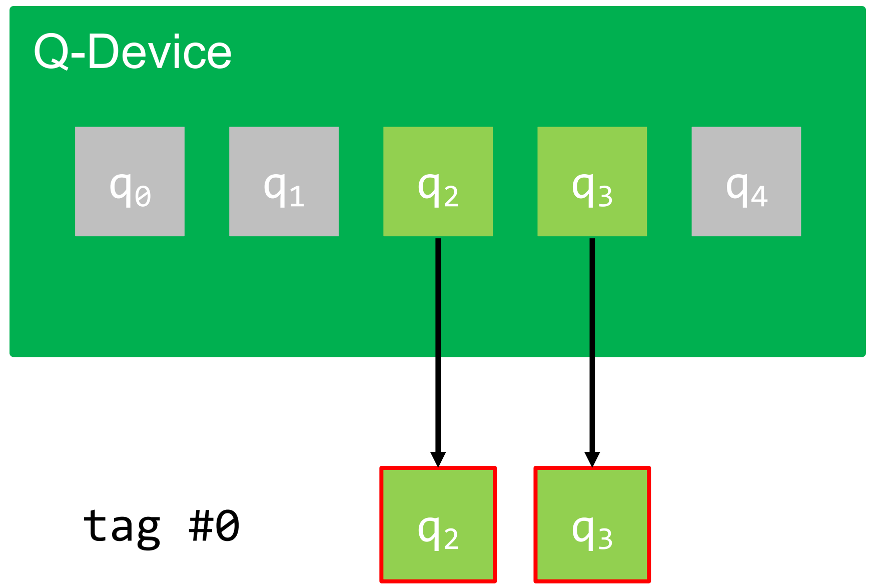
```
auto f0 = select<0,2,3>();
auto f1 = range<1,2>(f0);
```



Filters – views on the global Q-memory

- Starting from the full Q-memory filters restrict qubits step by step

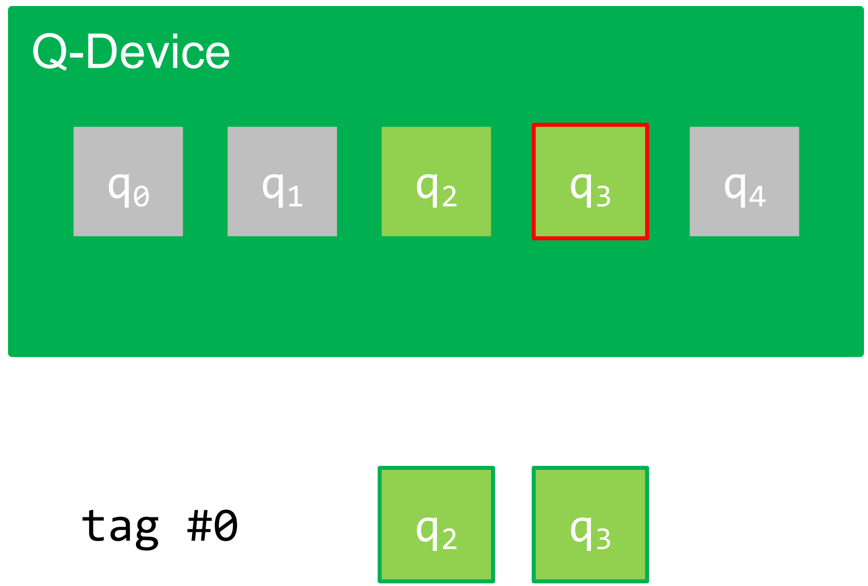
```
auto f0 = select<0,2,3>();
auto f1 = range<1,2>(f0);
auto f2 = tag<0>(f1);
```



Filters – views on the global Q-memory

- Starting from the full Q-memory filters restrict qubits step by step

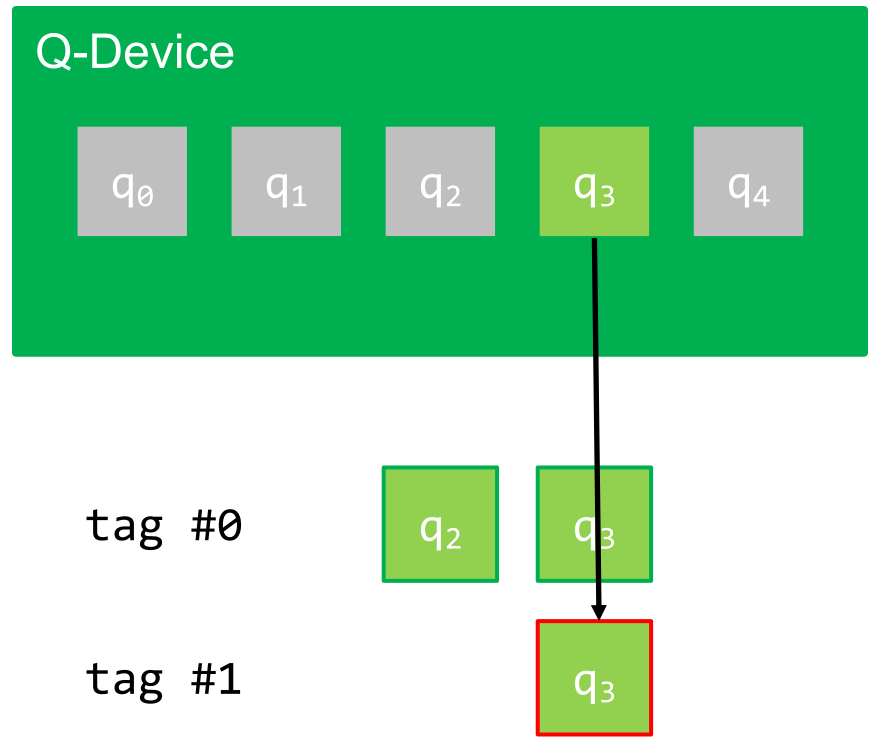
```
auto f0 = select<0,2,3>();
auto f1 = range<1,2>(f0);
auto f2 = tag<0>(f1);
auto f3 = qubit<1>(f2);
```



Filters – views on the global Q-memory

- Starting from the full Q-memory filters restrict qubits step by step

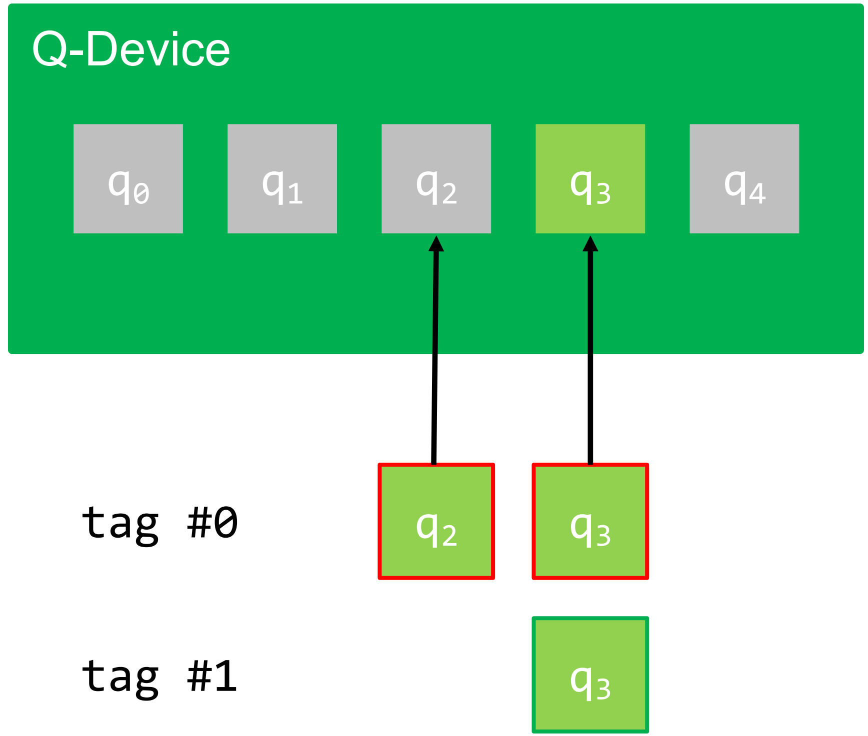
```
auto f0 = select<0,2,3>();
auto f1 = range<1,2>(f0);
auto f2 = tag<0>(f1);
auto f3 = qubit<1>(f2);
auto f4 = tag<1>(f3);
```



Filters – views on the global Q-memory

- Starting from the full Q-memory filters restrict qubits step by step

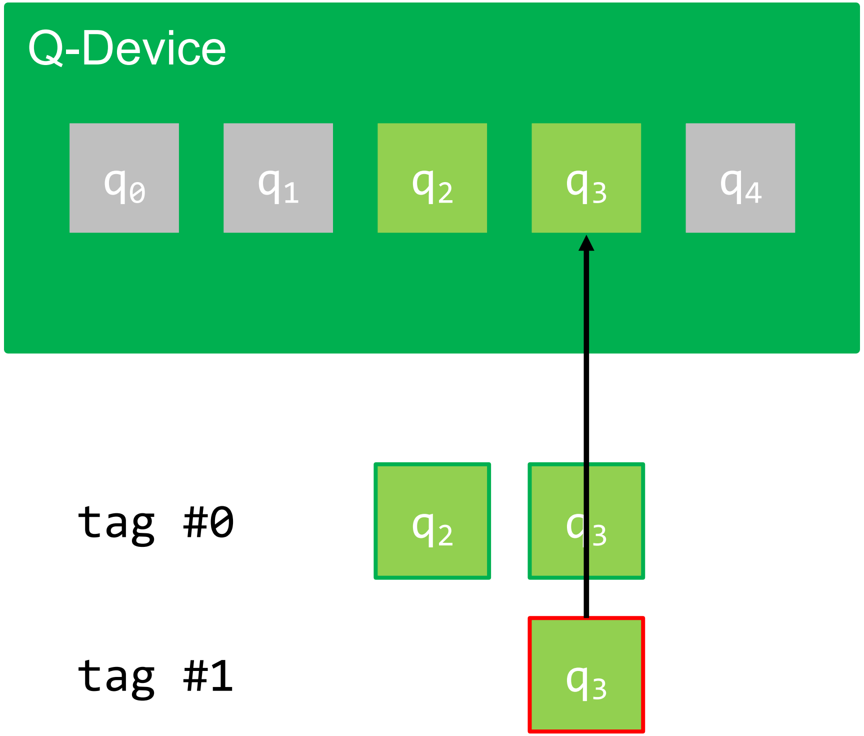
```
auto f0 = select<0,2,3>();
auto f1 = range<1,2>(f0);
auto f2 = tag<0>(f1);
auto f3 = qubit<1>(f2);
auto f4 = tag<1>(f3);
auto f5 = gototag<0>(f4);
```



Filters – views on the global Q-memory

- Starting from the full Q-memory filters restrict qubits step by step

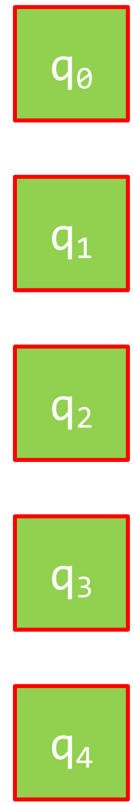
```
auto f0 = select<0,2,3>();
auto f1 = range<1,2>(f0);
auto f2 = tag<0>(f1);
auto f3 = qubit<1>(f2);
auto f4 = tag<1>(f3);
auto f5 = gototag<0>(f4);
auto f6 = gototag<1>(f5);
```



Gates – the salt of in-memory computing

- Gates apply to all qubits of the current filter chain (SIMD-ish)

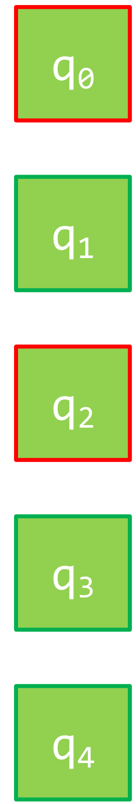
```
auto e0 = init();
```



Gates – the salt of in-memory computing

- Gates apply to all qubits of the current filter chain (SIMD-ish)

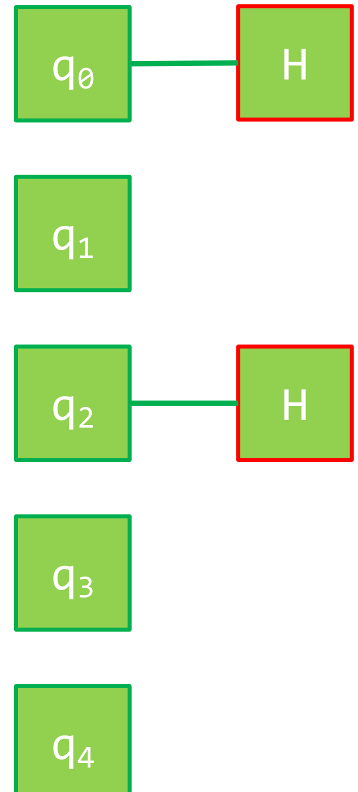
```
auto e0 = init();
auto e1 = sel<0,2>(e0);
```



Gates – the salt of in-memory computing

- Gates apply to all qubits of the current filter chain (SIMD-ish)

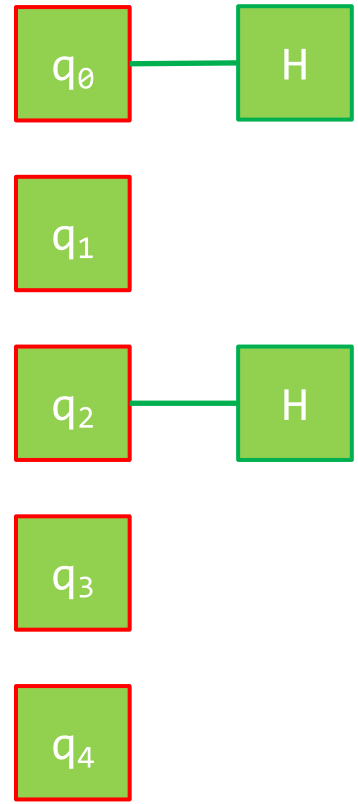
```
auto e0 = init();
auto e1 = sel<0,2>(e0);
auto e2 = h(e1);
```



Gates – the salt of in-memory computing

- Gates apply to all qubits of the current filter chain (SIMD-ish)

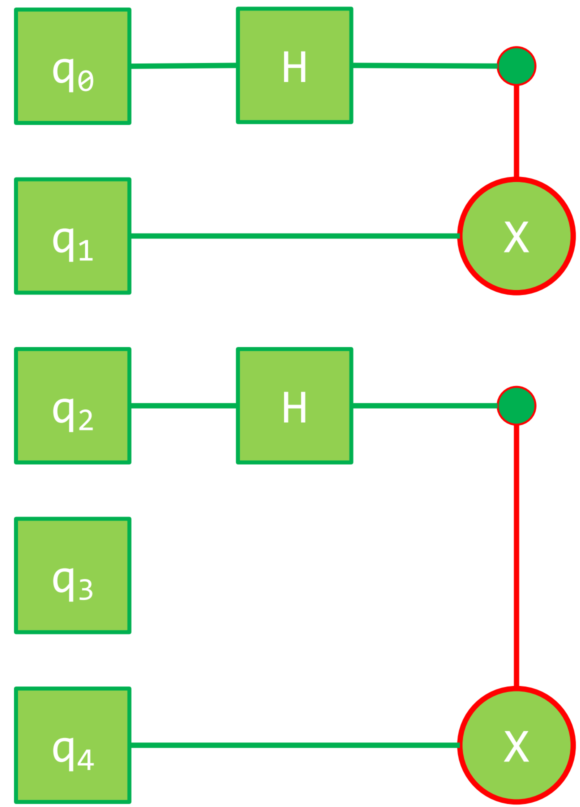
```
auto e0 = init();
auto e1 = sel<0,2>(e0);
auto e2 = h(e1);
auto e3 = all(e2);
```



Gates – the salt of in-memory computing

- Gates apply to all qubits of the current filter chain (SIMD-ish)

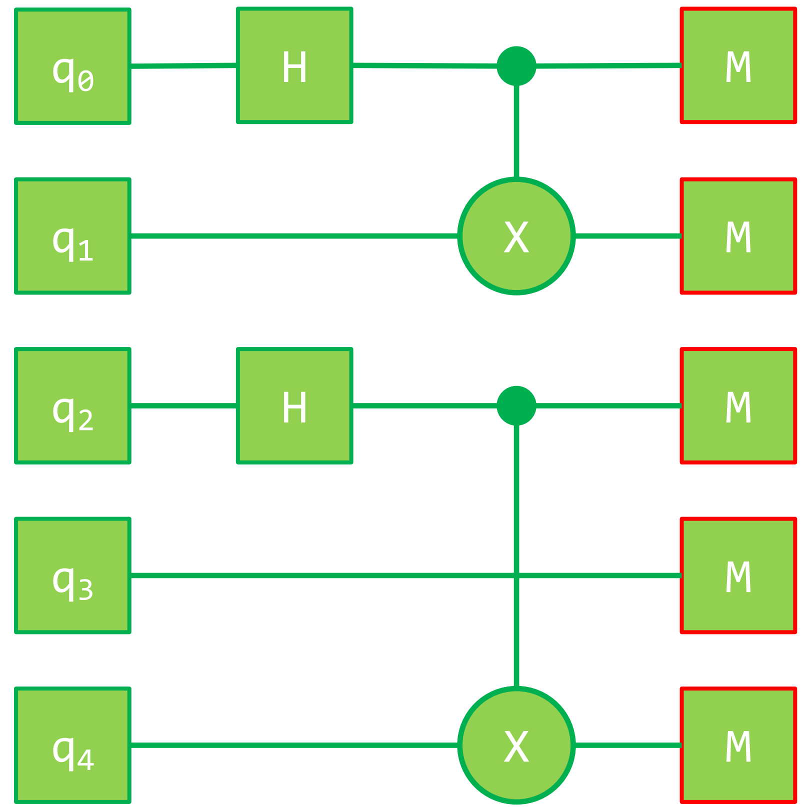
```
auto e0 = init();
auto e1 = sel<0,2>(e0);
auto e2 = h(e1);
auto e3 = all(e2);
auto e4 = cnot(
            sel<0,2>(),
            sel<1,4>(e3)
          );
```



Gates – the salt of in-memory computing

- Gates apply to all qubits of the current filter chain (SIMD-ish)

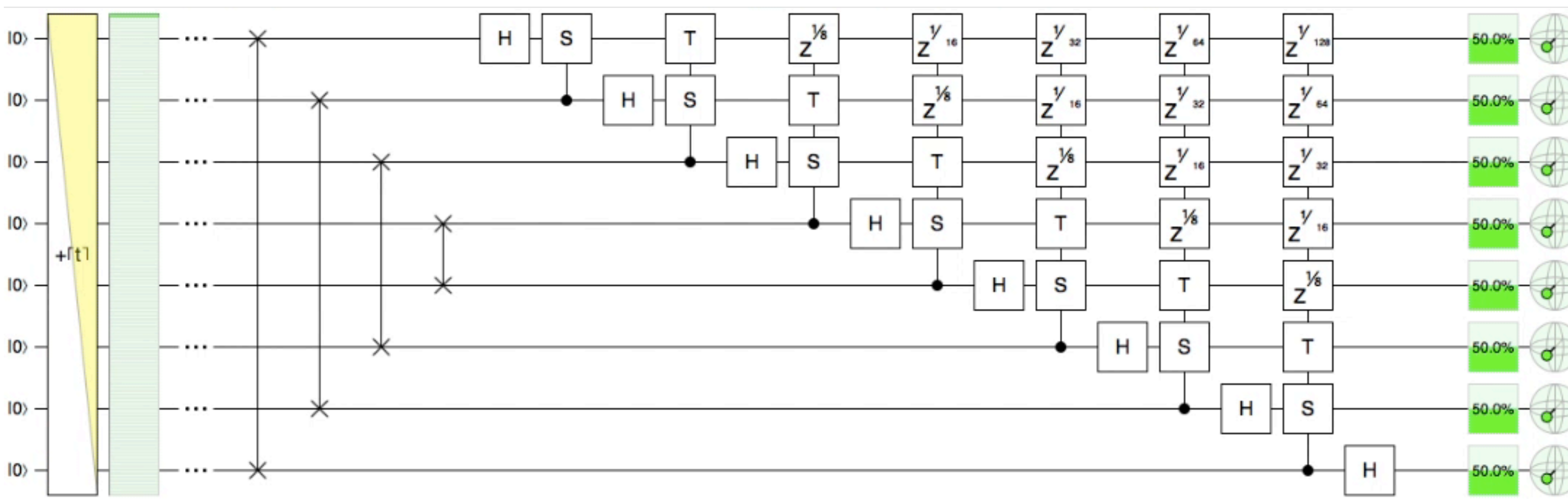
```
auto e0 = init();
auto e1 = sel<0,2>(e0);
auto e2 = h(e1);
auto e3 = all(e2);
auto e4 = cnot(
            sel<0,2>(),
            sel<1,4>(e3)
          );
auto e5 = measure(all(e4));
```



Circuits – pre-cooked quantum building blocks

- Generic quantum algorithms that can be applied to registers of arbitrary size

```
auto expr = qft(…);
```



Animation created with Quirk https://algassert.com/quirk

Rule-based optimization

- Unitarity of quantum gates

$$S \circ S^{\dagger} = S^{\dagger} \circ S = id$$

```
auto expr = s(sdag(…));
```

- Template metaprogramming

```
template<class Expr>
auto s(Expr&& expr)
{
    return QGate_S(expr);
}
```

Rule-based optimization

- Unitarity of quantum gates

$$S \circ S^{\dagger} = S^{\dagger} \circ S = id$$

```
auto expr = …;
```

- Template metaprogramming

```
template<class Expr>
auto s(Expr&& expr)
{
    return QGate_S(expr);
}
```

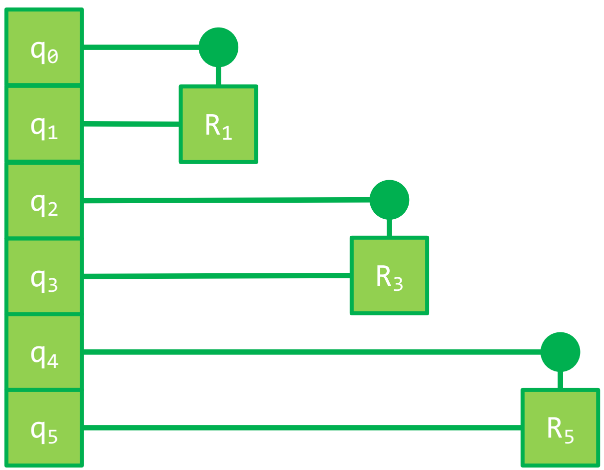
- Explicit template specialization

```
template<>
auto s(QGate_Sdag&& expr)
{
  return expr.getSubexpr();
}
```

Compile-time loops

- For-loop call

```
auto expr =
static_for<1,5,2,body>(…);
```



- For-loop body

```
struct body
{
  template<size_t k,
           class  Expr>
  static constexpr auto
  func(Expr&& expr) noexcept
  {
    return crk<k>(
      sel<k-1>(all( )),
      sel<k  >(all(expr)));
  }
};
```

Advanced techniques

- Hook gate for user-defined mini-circuits
- Just-in-time compilation of run-time generated quantum expressions

Work in progress

- Decomposition gates, e.g. $U = R_z(\varphi_1)R_y(\varphi_2)R_z(\varphi_3)$
- QInteger and QPosit arithmetics
- C and Python API using JIT compilation

FPGA-ish ‘synthesis’

- Generic quantum expression

  ```
  auto expr = qft(init());
  ```

  is independent of
  - Q-device type
  - Q-memory size (#qubits)
  - concrete input data

FPGA-ish 'synthesis'

- Generic quantum expression

  ```
  auto expr = qft(init());
  ```

  is independent of
  - Q-device type
  - Q-memory size (#qubits)
  - concrete input data

- Q-device specific kernel code

  ```
  QData<6, cQASMv1> data;
  cout << expr(data);
  ```

```
version 1.0
qubits 6
h q[0]
cr q[1], q[0], 1.570796326794896558
cr q[2], q[0], 0.785398163397448279
cr q[3], q[0], 0.392699081698724139
cr q[4], q[0], 0.196349540849362070
cr q[5], q[0], 0.098174770424681035
h q[1]
cr q[2], q[1], 1.570796326794896558
cr q[3], q[1], 0.785398163397448279
cr q[4], q[1], 0.392699081698724139
cr q[5], q[1], 0.196349540849362070
h q[2]
cr q[3], q[2], 1.570796326794896558
cr q[4], q[2], 0.785398163397448279
cr q[5], q[2], 0.392699081698724139
h q[3]
cr q[4], q[3], 1.570796326794896558
cr q[5], q[3], 0.785398163397448279
h q[4]
cr q[5], q[4], 1.570796326794896558
h q[5]
swap q[0], q[5]
swap q[1], q[4]
swap q[2], q[3]
```

Quantum-Inspire

FPGA-ish ‘synthesis’

- Generic quantum expression

  ```
  auto expr = qft(init());
  ```

  is independent of
  - Q-device type
  - Q-memory size (#qubits)
  - concrete input data

- Q-device specific kernel code

  ```
  QData<6, openQASMv2> data;
  cout << expr(data);
  ```

```
version 1.0
qubits 6
h q[0]
cr q[1], q[0], 1.570796326794896558
cr q[2], q[0], 0.785398163397448279
cr q[3], q[0], 0.392699081698724139
cr q[4], q[0], 0.196349540849362070
cr q[5], q[0], 0.098174770424681035
h q[1]
cr q[2], q[1], 1.570796326794896558
cr q[3], q[1], 0.785398163397448279
cr q[4], q[1], 0.392699081698724139
cr q[5], q[1], 0.196349540849362070
h q[2]
cr q[3], q[2], 1.570796326794896558
cr q[4], q[2], 0.785398163397448279
cr q[5], q[2], 0.392699081698724139
h q[3]
cr q[4], q[3], 1.570796326794896558
cr q[5], q[3], 0.785398163397448279
h q[4]
cr q[5], q[4], 1.570796326794896558
h q[5]
swap q[0], q[5]
swap q[1], q[4]
swap q[2], q[3]
```

Quantum-Inspire

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[6];
creg c[6];
h q[0];
cu1(1.570796326794896558) q[1], q[0];
cu1(0.785398163397448279) q[2], q[0];
cu1(0.392699081698724139) q[3], q[0];
cu1(0.196349540849362070) q[4], q[0];
cu1(0.098174770424681035) q[5], q[0];
h q[1];
cu1(1.570796326794896558) q[2], q[1];
cu1(0.785398163397448279) q[3], q[1];
cu1(0.392699081698724139) q[4], q[1];
cu1(0.196349540849362070) q[5], q[1];
h q[2];
cu1(1.570796326794896558) q[3], q[2];
cu1(0.785398163397448279) q[4], q[2];
cu1(0.392699081698724139) q[5], q[2];
h q[3];
cu1(1.570796326794896558) q[4], q[3];
cu1(0.785398163397448279) q[5], q[3];
h q[4];
cu1(1.570796326794896558) q[5], q[4];
h q[5];
swap q[0], q[5];
swap q[1], q[4];
swap q[2], q[3];
```

IBM Q Experience

CUDA-ish stream execution model

- High latency is caused by
  - Python-based vendor tools and complexity of the process
  - remote access to cloud-based Q-devices with waiting queues

```
// Blocking execution
QJob* job = data.execute(…);

// Result as JSON object
json result = job->get();
```

CUDA-ish stream execution model

- High latency is caused by
  - Python-based vendor tools and complexity of the process
  - remote access to cloud-based Q-devices with waiting queues
- Asynchronous execution
  - hides latencies by continuing the classical program flow

```
// Non-blocking execution
QJob* job = data.execute_async(…);

// do other tasks

// Wait for completion
job->wait();
```

CUDA-ish stream execution model

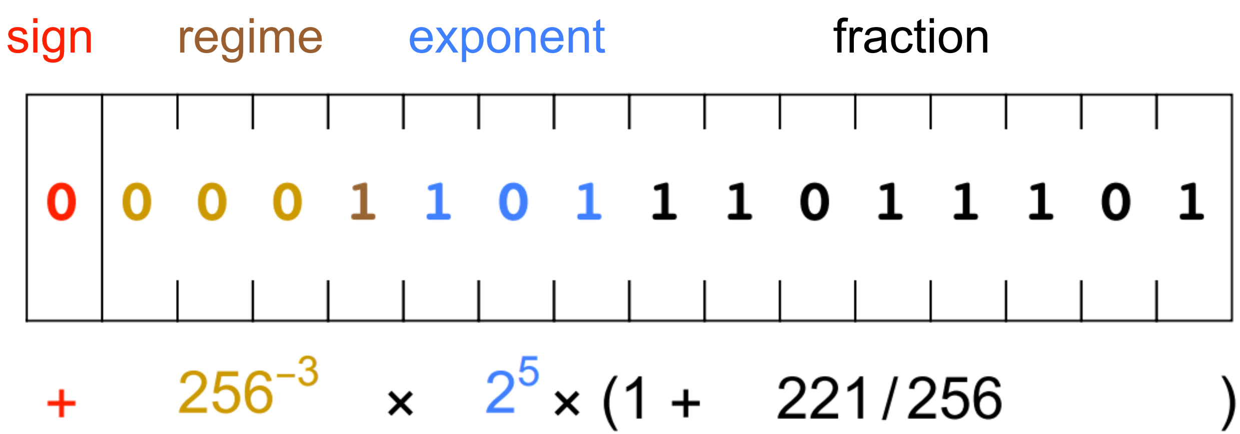
- High latency is caused by
  - Python-based vendor tools and complexity of the process
  - remote access to cloud-based Q-devices with waiting queues
- Asynchronous execution
  - hides latencies by continuing the classical program flow
  - enables concurrent execution of kernels via multiple streams

```
QStream stream0, stream1;

QJob* job0 =
  data0.execute_async(stream0,…);
QJob* job1 =
  data1.execute_async(stream1,…);

// do other tasks

if (job0->query()) { … }
if (job1->query()) { … }
```

ONGOING DEVELOPMENTS

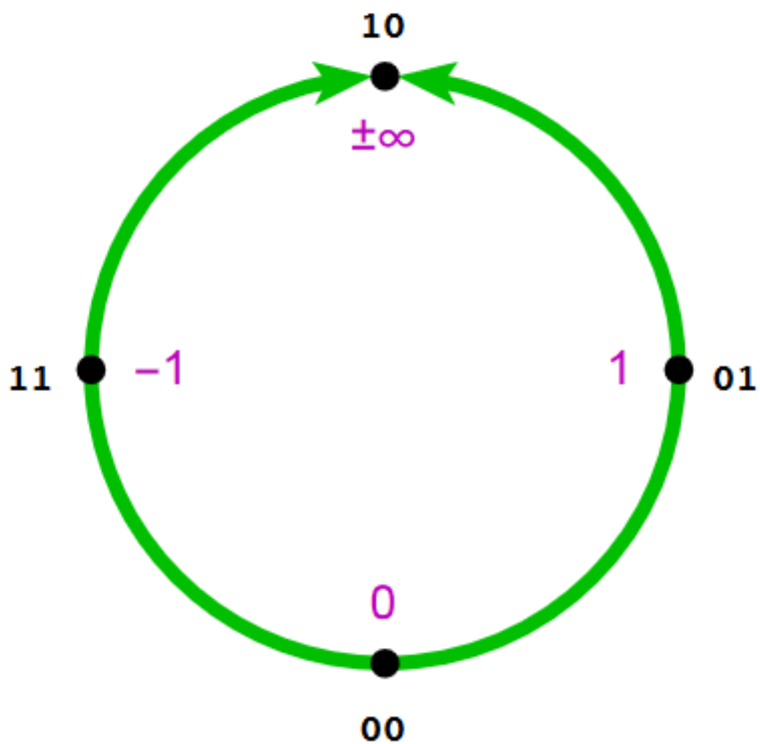
Real-valued data

- IEEE-754 floating points require 32-64 qubits per datum → impractical
- Encoding real-number in a single qubit → tempting but not succeeded yet
- More (qu)bit efficient number formats → Posits (Type III UNUMs)

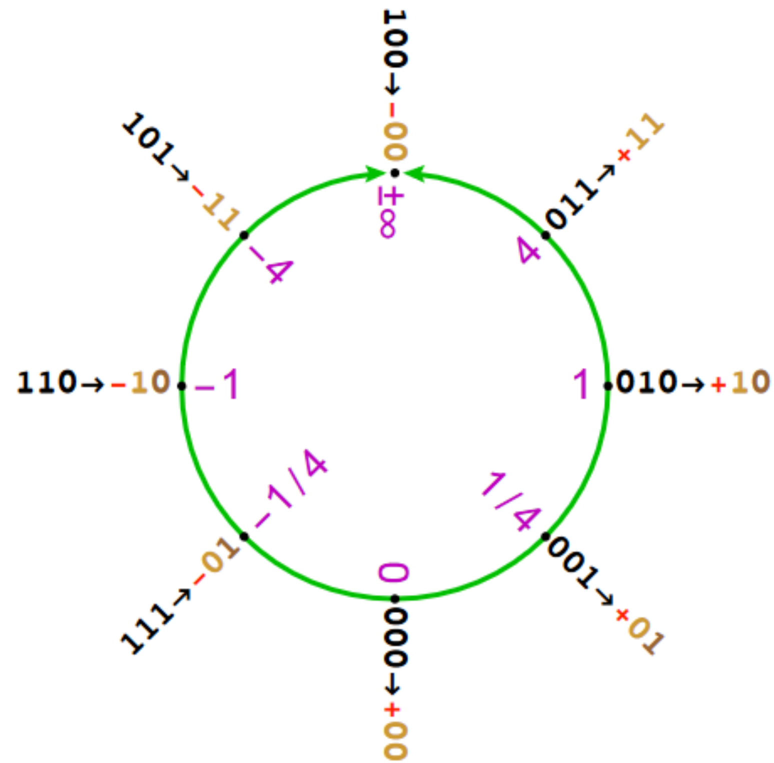


J. Gustafson: Beating floating point at its own game: posit arithmetic

Posits

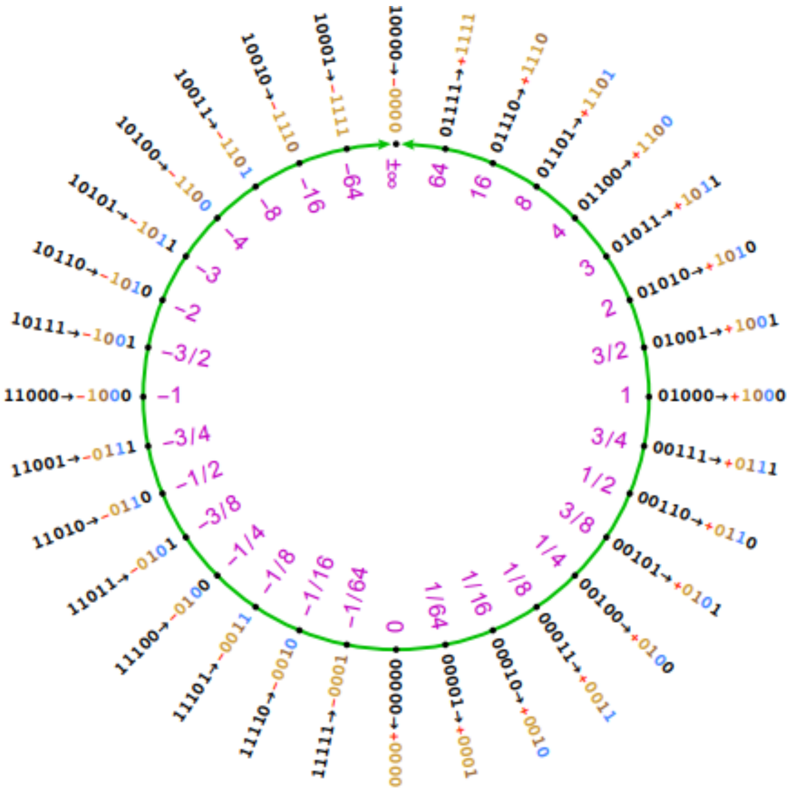


10
±∞
11
−1
1
01
0
00



100→−00
±∞
101→−11
−4
011→+11
4
110→−10
−1
1
010→+10
111→−01
−1/4
1/4
001→+01
0
000→+00

Posit arithmetic



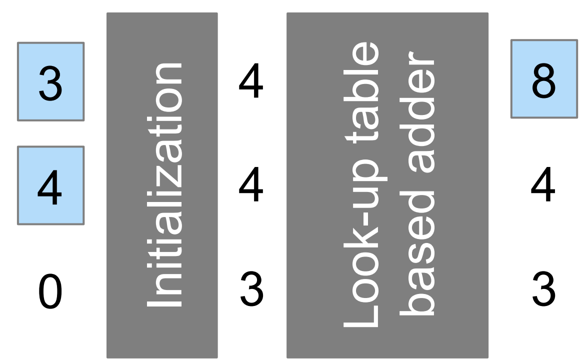
- Example:

3 = +1011

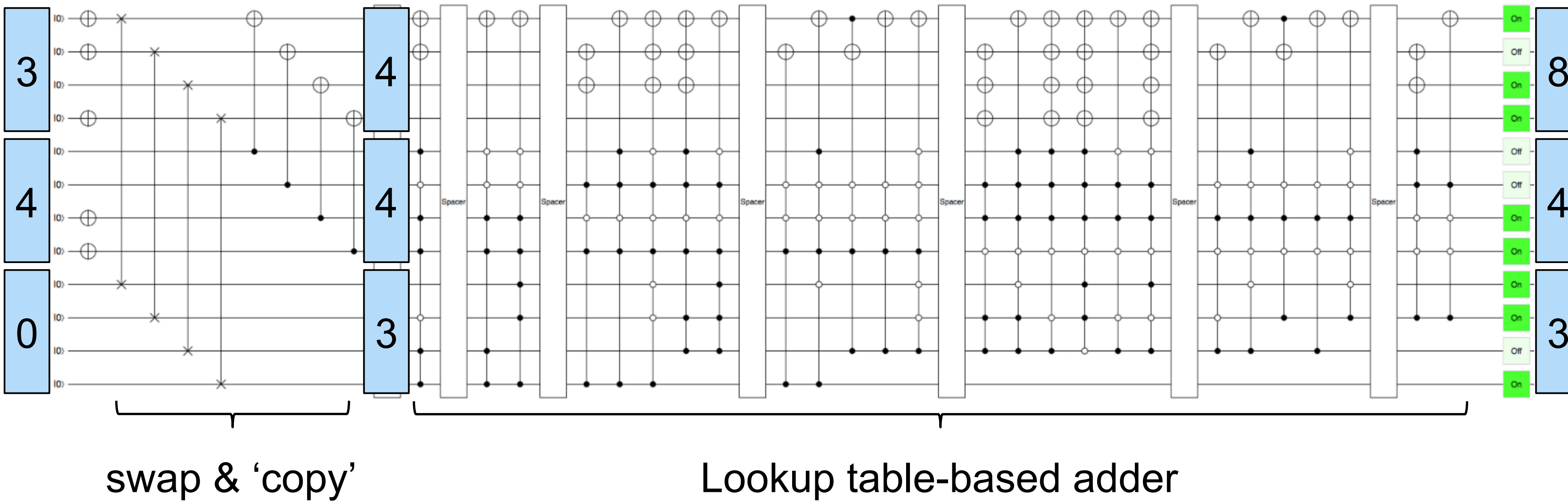
4 = +1100

---

8 = +1101

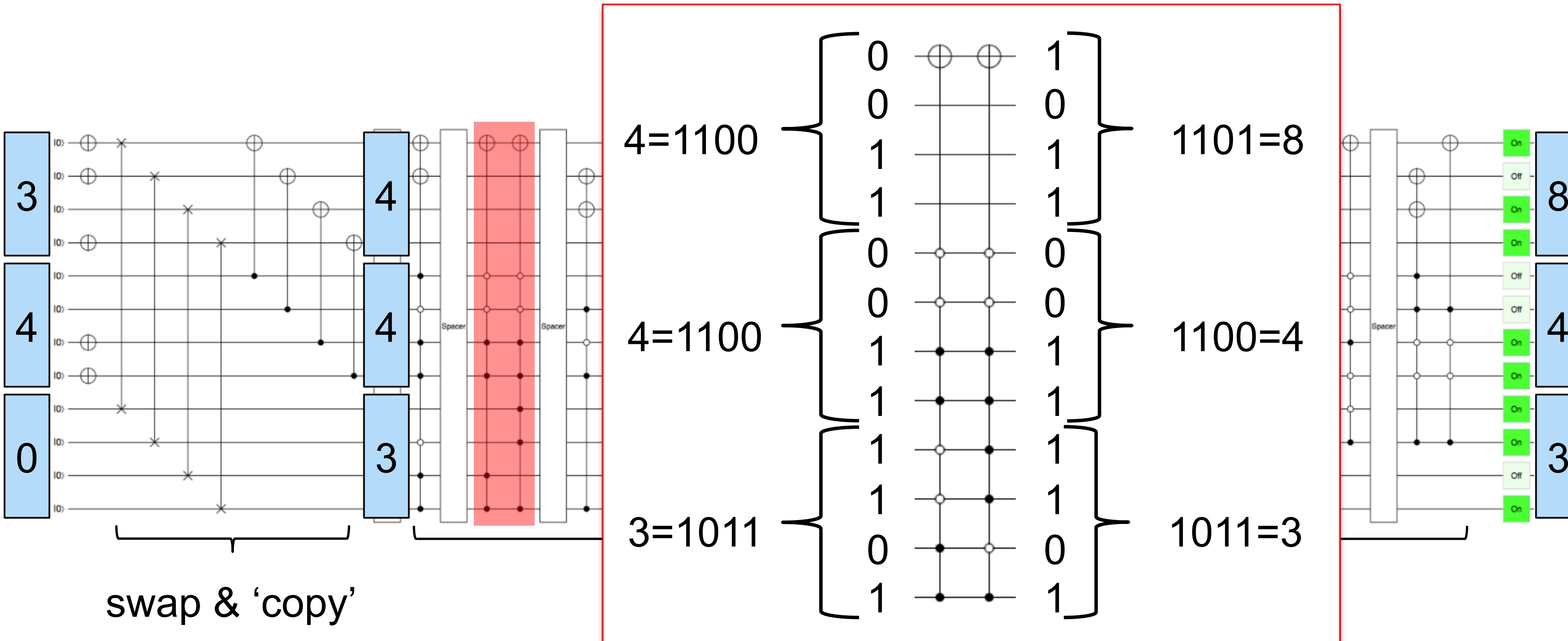


Posit arithmetic on quantum computers



T. Driebergen: Designing a Quantum Algorithm for Real-Valued Addition Using Posit Arithmetic, BSc Thesis, TU Delft, 2019

Posit arithmetic on quantum computers



T. Driebergen: Designing a Quantum Algorithm for Real-Valued Addition Using Posit Arithmetic, BSc Thesis, TU Delft, 2019

Conclusion

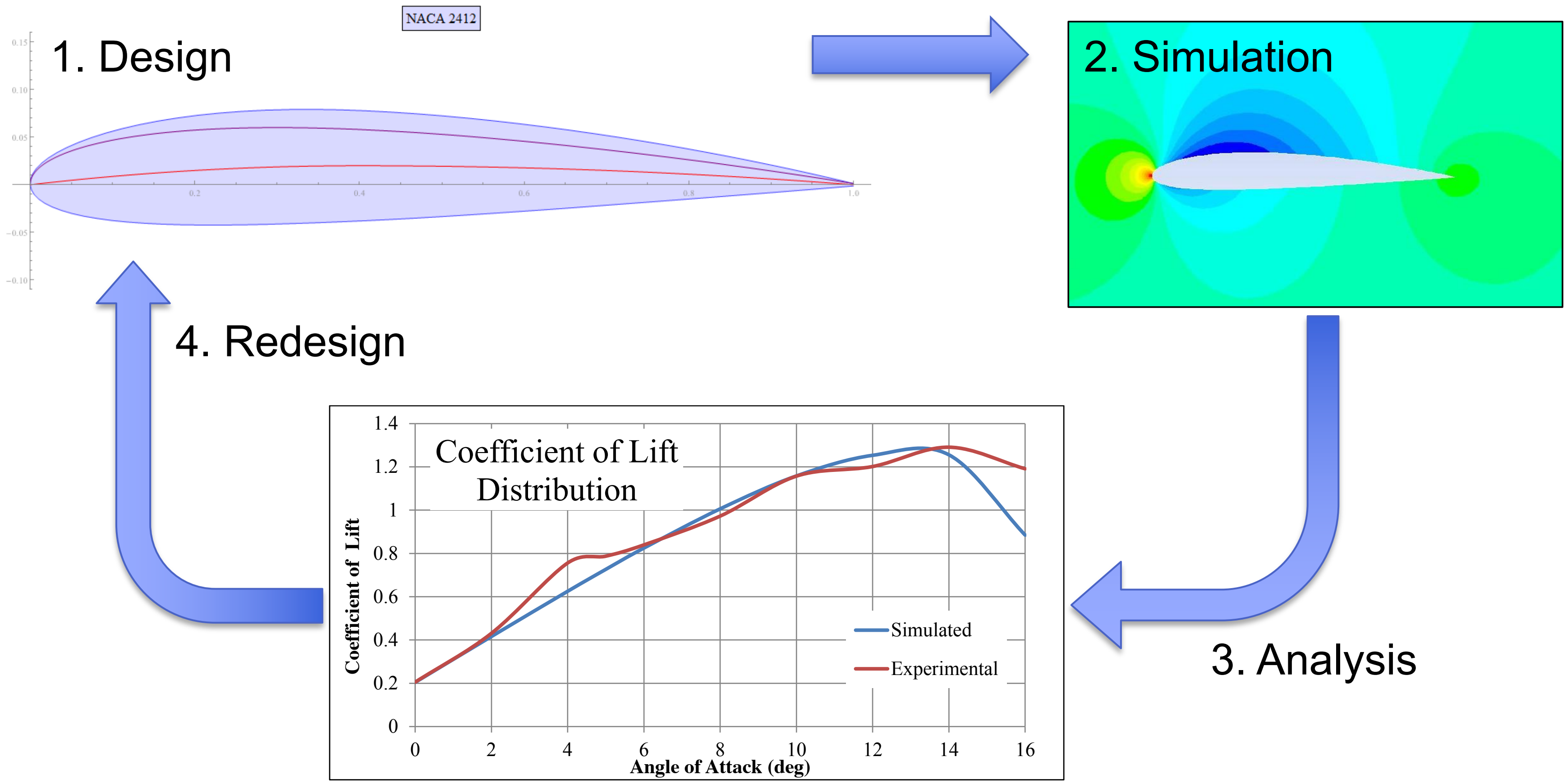
- **A cross-platform SDK for Q-accelerated scientific computing**
  - Rapid prototyping and testing of quantum expressions
  - Seamless integration into (C-accelerated) applications

- **Ongoing work**
  - Implementation of HHL and QInteger/QPosit arithmetics
  - Cloud platform https://INGInious.ewi.tudelft.nl

- **Publications**
  - MM, Schalkers: A cross-platform programming framework for quantum-accelerated scientific computing. Submitted to ICCS 2020
  - Driebergen, MM: A novel quantum algorithm for adding real-valued numbers using posit arithmetic. Submitted to RC 2020

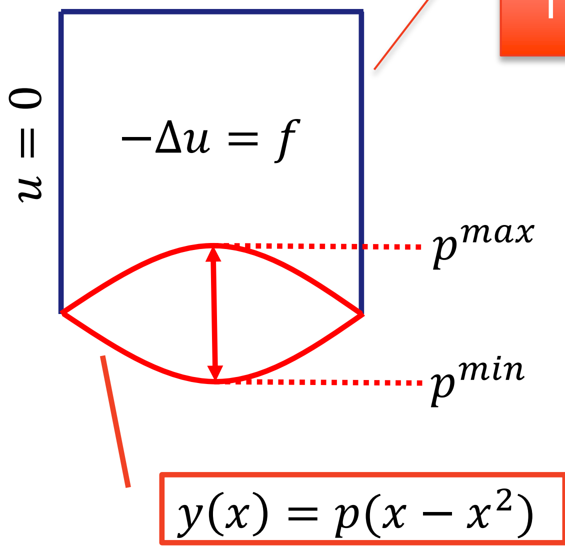
Extra Slides

Simulation-based design and analysis cycle



Matsson et al. Aerodynamic Performance of the NACA 2412 Airfoil at Low Reynolds Number, 2016 ASEE Annual Conference & Exposition

Academic model problem



1. Design

4. Redesign

- **Problem:** Minimize the difference

$$d_h = u_h - u_h^*$$

between the solution u_h and a given profile u_h^* w.r.t.

3. Analysis

$$\mathcal{C}(d_h, p) = d_h^T M d_h$$

such that d_h solves

2. Simulation

$$A_h d_h = f_h - A_h u_h^*$$