Introduction to Quantum Computing & Hybrid HPC-QC Systems

SW stack

Miroslav Dobsicek

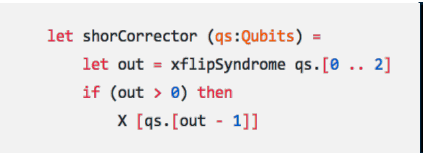
NordIQuEst WACQT ENCCS

Presentation overview

- SW stack overview
- Circuit level assembly
- Hardware level encoding
- LUMI - QAL9000 PoC connection

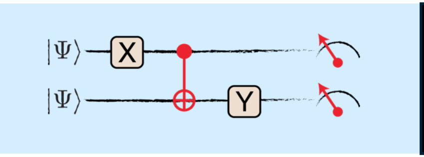
SW stack overview

SW stack

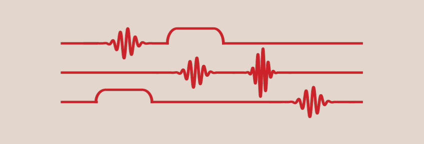


```
let shorCorrector (qs:Qubits) =
    let out = xflipSyndrome qs.[0 .. 2]
    if (out > 0) then
        X [qs.[out - 1]]
```

Circuit design



Compiler



Pulse schedules

Instrument orchestration

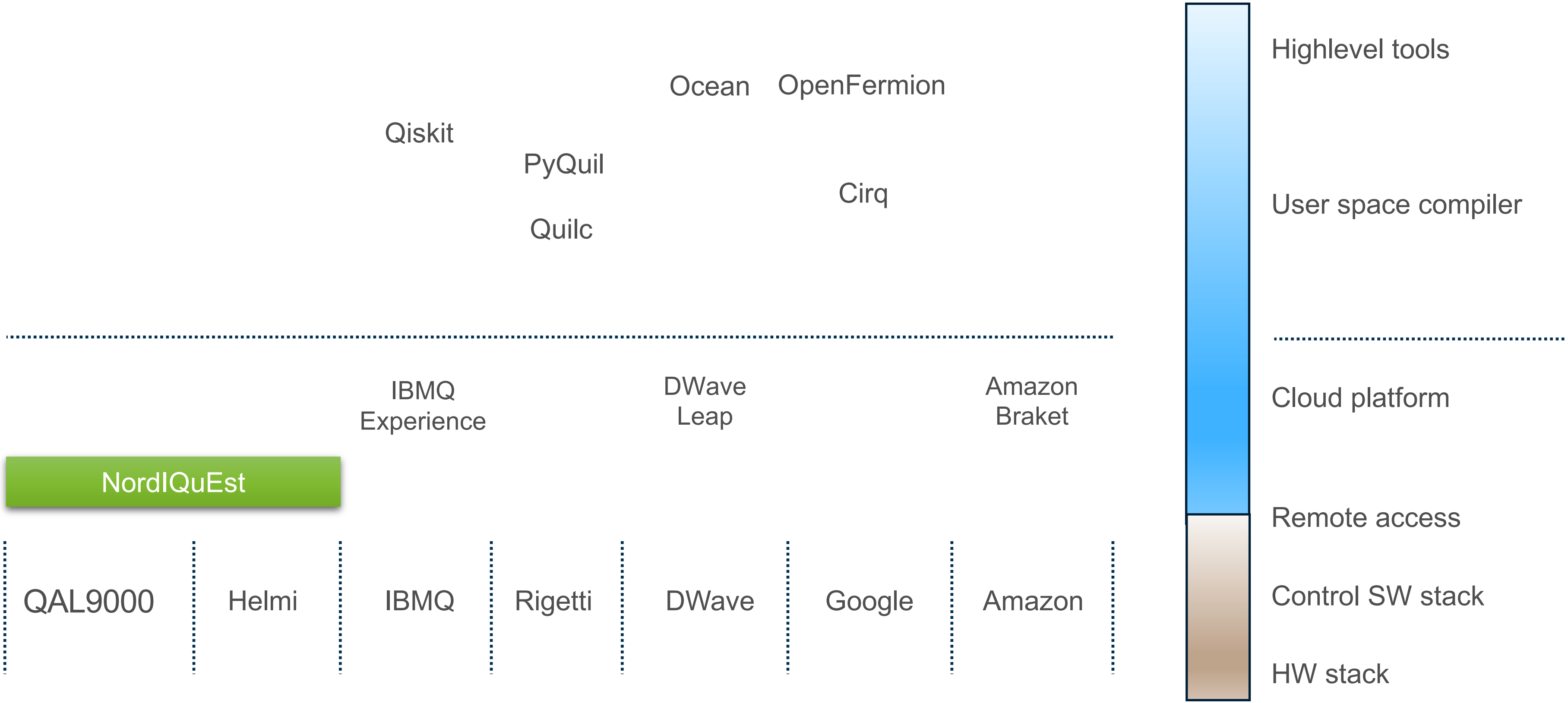
Computer science domain
Output for idealized quantum computer

Co-design for NISQ devices

Experimentalist domain
Single-user environment, lab work

Many available platforms

OpenQASMv2 is an interoperability workhorse



Ocean

OpenFermion

Qiskit

PyQuil

Cirq

Quilc

IBMQ Experience

DWave Leap

Amazon Braket

NordIQuEst

QAL9000

Helmi

IBMQ

Rigetti

DWave

Google

Amazon

Highlevel tools

User space compiler

Cloud platform

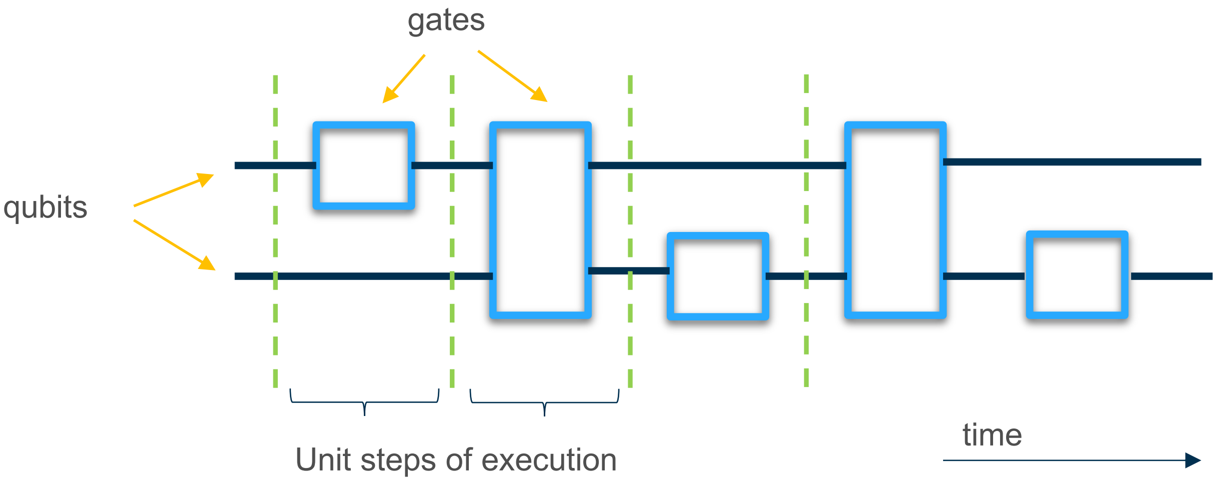
Remote access

Control SW stack

HW stack

Circuit based quantum computing model

- This model is pretty central to most current quantum computing architectures
- SW stack grows both above and below it.



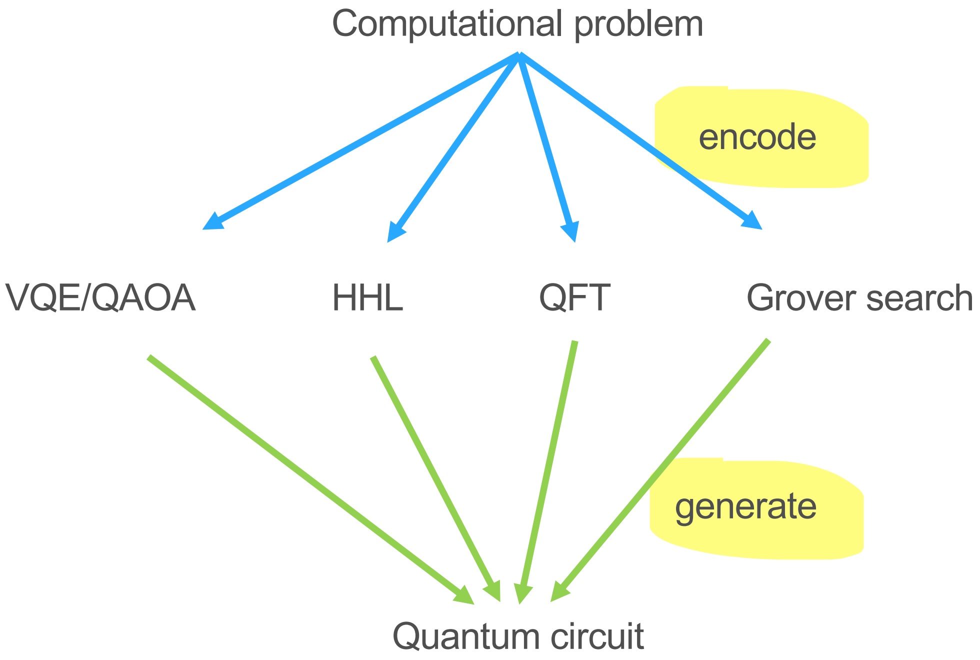
- In classical logical circuits, it is an electric signal which is being propagated through position-fixed gates.
- With superconducting qubits, it is the othe way around. Qubits sit still and gates are applied step-wise to them in a form of mw-pulses.

High level parts of a SW stack

How do we generate quantum circuits?

1. **Encode** your problem into known generic (speedup) quantum algorithms
2. **Embed** a classical circuit in a quantum one via reversible logic
3. Try **automatic decomposition** of large transformations into sequences of smaller ones
4. **Design** a novel quantum algorithm

1. Problem enconding into an existing quantum algorithm



Computational problem

encode

VQE/QAOA

HHL

QFT

Grover search

generate

Quantum circuit

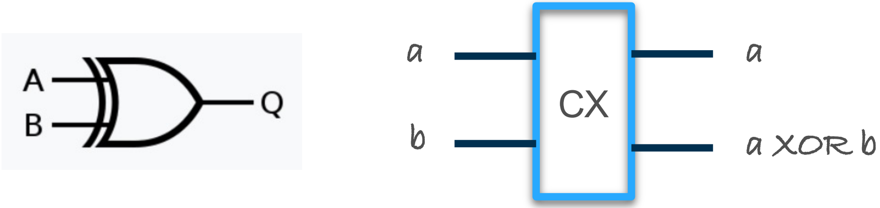
This is currently the most feasible way how to do a computation on a quantum computer.

- VQE – quantum chemistry problems
- QAOA – combinatorial opt. problems
- HHL – systems of linear equations (ML)
- QFT – detect group-like properties
- Grover search – generic square root speed-up

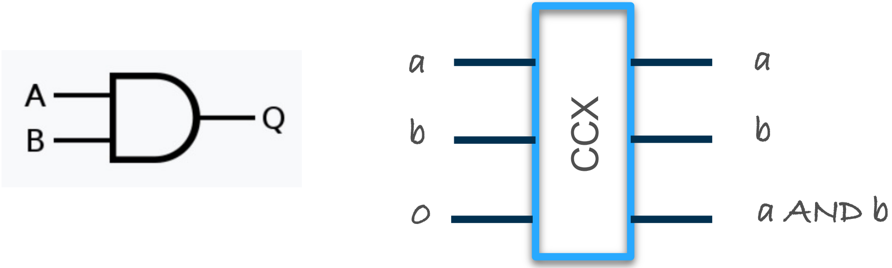
2. Embedding of classical circuits via reversible logic

NOT

XOR



AND



Classical logical gates mostly map to quantum gates in 1:1 fashion.

A quantum circuit generated in this way will have the same overall complexity as the classical circuit. Not better or worse. But! it will be capable of working with superposition of states. The cost are extra qubits guaranteeing reversibility.

Do you know that the QFT circuit and the circuit for a classical FFT are the same?

3. Automatic decomposition

Desired transformation:

$$\mathrm{QFT}: |x\rangle \mapsto \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \omega_N^{xk} |k\rangle .$$

You start with a mathematical description of the desired unitary transformation. Expansion to a matrix form is straightforward. Then apply unitary decomposition algorithm(s). This process is usually based on Singular Value Decomposition (SVD).

Matrix form:

$$F_N = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

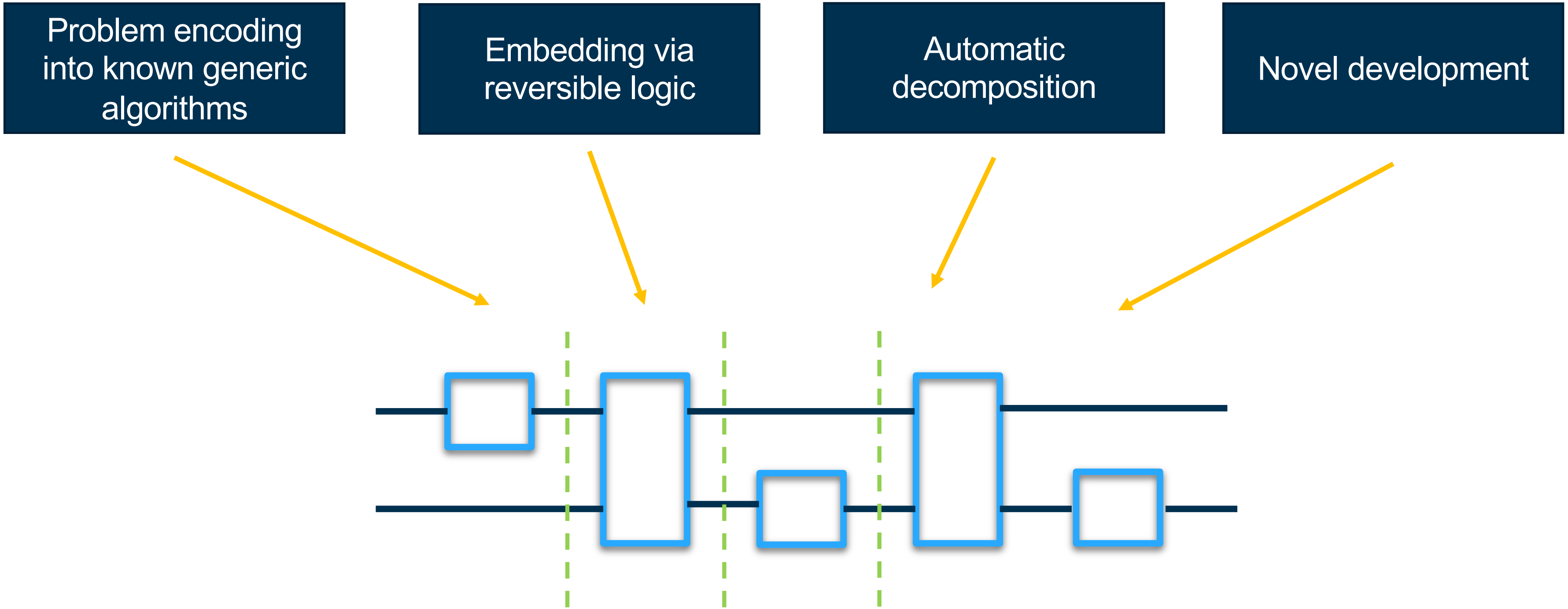
This approach is unlikely to lead to efficent circuits! The number of generated gates is exponential in the number of qubits in general.

However, unitary decomposition is very useful when breaking smaller gates into a specific target gate set.

4. Novel design

- Not an easy task
- Much of reasoning is still tied to circuits and complex Hilbert spaces
- We are “chasing vectors around” in an analogy to “chasing bits around”
- The most active fields in quantum algorithm theory are:
    - Quantum error correction codes
    - Quantum complexity classes
        - MIP* = RE, Certifiable randomness, Classically verifiable quantum advantage
    - Finding new classical algorithms by ”dequantization”

Summary of circuit generation



Problem encoding into known generic algorithms

Embedding via reversible logic

Automatic decomposition

Novel development

Circuit level assembly

A number of circuit optimizations

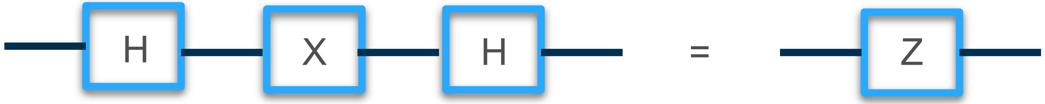
1. **Circuit compression** – minimize the number of gates used (coupling gates in particular)
2. **Unroll/decompose** to the native gate set supported by the quantum HW
3. **Optimal routing** – map the logical circuit to the physical chip while respecting its connectivity map. Insert SWAP gates where needed.
4. Insert **error mitigation** gates.

These optimizations techniques are partwise orthogonal, quantum HW dependent, and may be applied iteratively/recursively in order to achieve best results.

Circuit compression

- The most common technique is to exploit logical circuit identities

Eg:

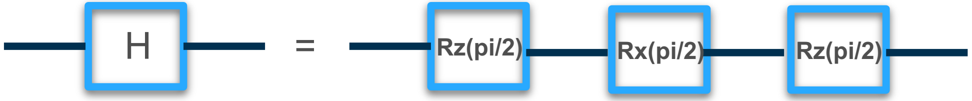


arXiv:2012.13966

- One of the newer approaches is called **ZX-calculus**.
  - It relaxes the unitarity condition: operating in a less restrictive linear regime instead.
  - But, it's not always possible to revert back to a unitary circuit.

Unrolling/decomposition

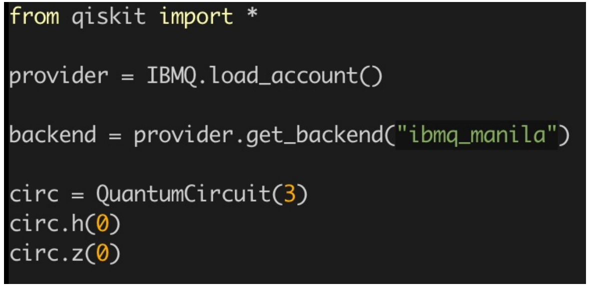
- There are many universal gate sets for quantum computing.
- For superconducting qubits the set is often composed of entangling gates **CX**, **CZ**, or **iSWAP** accopanied with **Rx(..)** and **Rz(..)** single qubit rotations. We call it a **native** gate set.
- SW stack typically contains a **library** of definitions of other **commons gates** in terms of the native universal gate set. Thus, for example, the Hadamard gate H can be 'unrolled' in terms of Rx(..) and Rz(..).



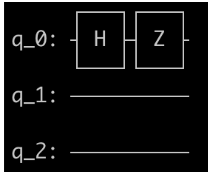
- Uncommon gates needs to be (brute-force) decomposed (SVD).

Qiskit contains plentyful of built-in circuit optimizations

Original circuit

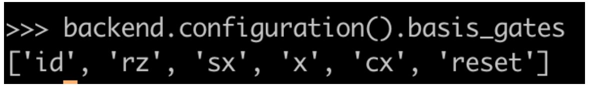


```
from qiskit import *

provider = IBMQ.load_account()

backend = provider.get_backend("ibmq_manila")

circ = QuantumCircuit(3)
circ.h(0)
circ.z(0)
```



```
     ┌───┐┌───┐
q_0: ┤ H ├┤ Z ├
     └───┘└───┘
q_1: ──────────

q_2: ──────────
```

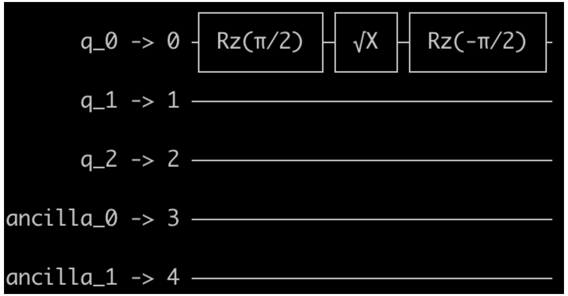
Check native gate set



```
>>> backend.configuration().basis_gates
['id', 'rz', 'sx', 'x', 'cx', 'reset']
```

Transpiled circuit

```
trc = transpile(circ, backend)
```

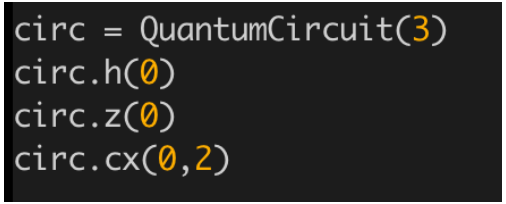


```
               ┌─────────┐┌────┐┌──────────┐
      q_0 -> 0 ┤ Rz(π/2) ├┤ √X ├┤ Rz(-π/2) ├
               └─────────┘└────┘└──────────┘
      q_1 -> 1 ─────────────────────────────

      q_2 -> 2 ─────────────────────────────

ancilla_0 -> 3 ─────────────────────────────

ancilla_1 -> 4 ─────────────────────────────
```

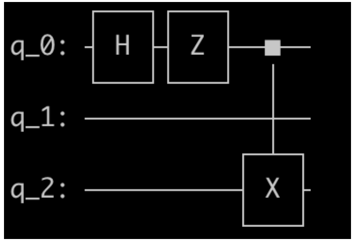
Unrolling and compression has been applied.

Qiskit contains plentyful of built-in circuit optimizations

Original circuit

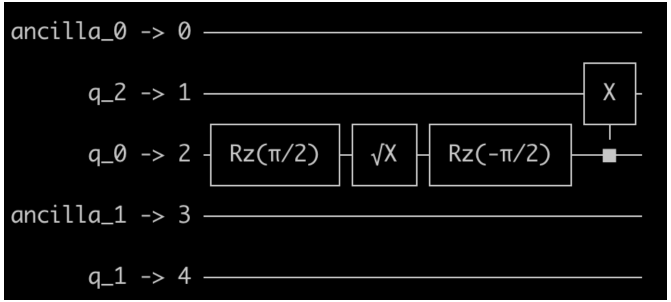


```
circ = QuantumCircuit(3)
circ.h(0)
circ.z(0)
circ.cx(0,2)
```



Transpiled circuit

```
trc = transpile(circ, backend)
```

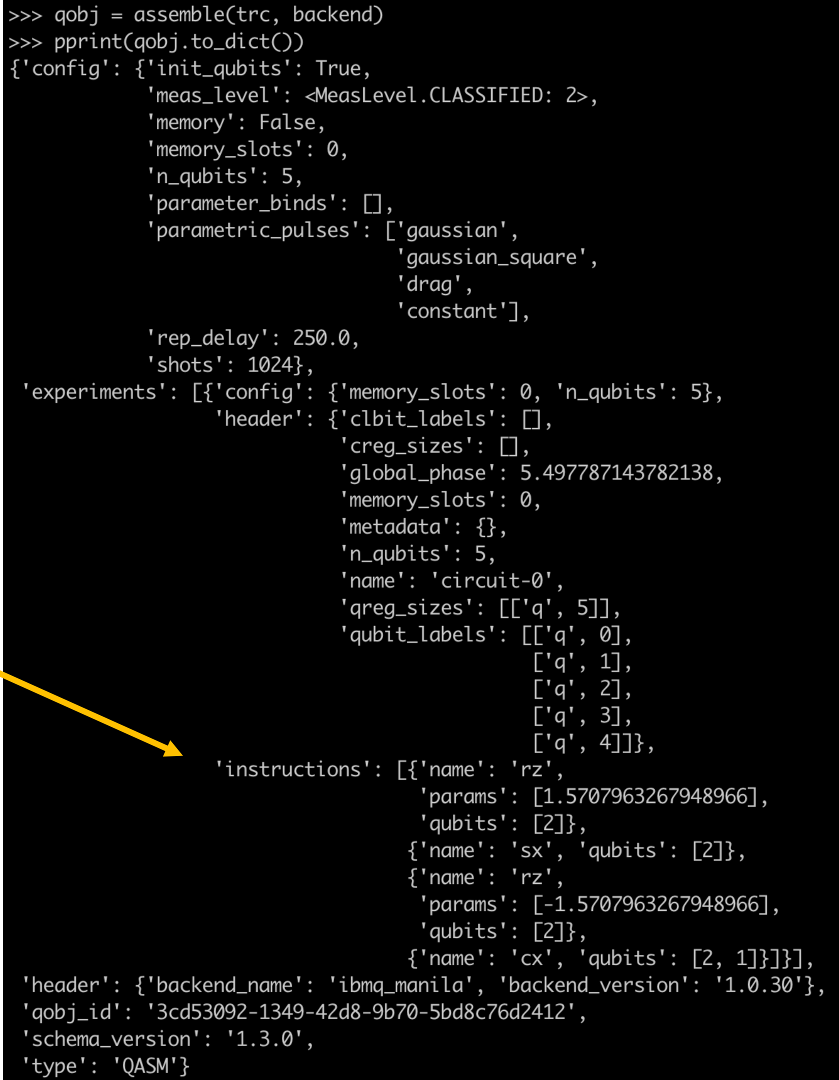


Manila’s coupling map

Unrolling, compression
and routing has been applied.

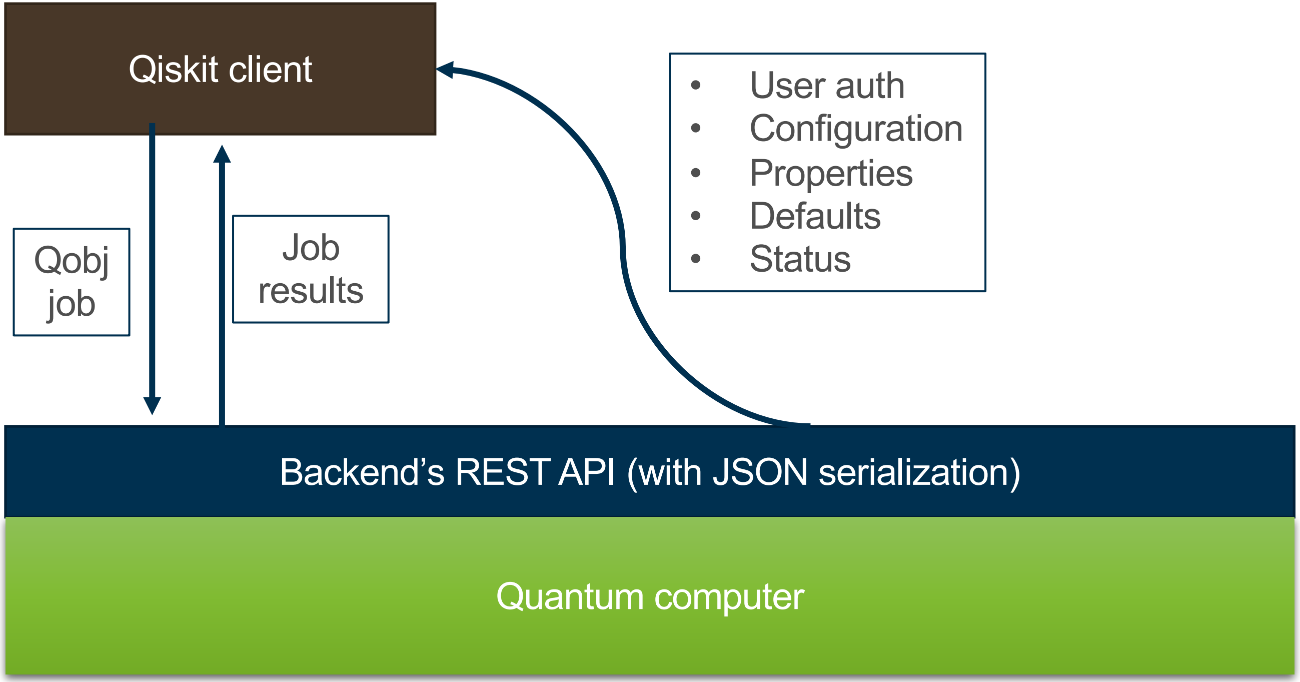
Finally, we can construct an assembly suitable for a remote execution

- In Qiskit lingo, this assembly is called Qobj.
- It is simply a serialization of the circuit into a textual form: see the **“instructions”** field,
- Plus metadata.



```
>>> qobj = assemble(trc, backend)
>>> pprint(qobj.to_dict())
{'config': {'init_qubits': True,
            'meas_level': <MeasLevel.CLASSIFIED: 2>,
            'memory': False,
            'memory_slots': 0,
            'n_qubits': 5,
            'parameter_binds': [],
            'parametric_pulses': ['gaussian',
                                  'gaussian_square',
                                  'drag',
                                  'constant'],
            'rep_delay': 250.0,
            'shots': 1024},
 'experiments': [{'config': {'memory_slots': 0, 'n_qubits': 5},
                  'header': {'clbit_labels': [],
                             'creg_sizes': [],
                             'global_phase': 5.497787143782138,
                             'memory_slots': 0,
                             'metadata': {},
                             'n_qubits': 5,
                             'name': 'circuit-0',
                             'qreg_sizes': [['q', 5]],
                             'qubit_labels': [['q', 0],
                                              ['q', 1],
                                              ['q', 2],
                                              ['q', 3],
                                              ['q', 4]]},
                  'instructions': [{'name': 'rz',
                                    'params': [1.5707963267948966],
                                    'qubits': [2]},
                                   {'name': 'sx', 'qubits': [2]},
                                   {'name': 'rz',
                                    'params': [-1.5707963267948966],
                                    'qubits': [2]},
                                   {'name': 'cx', 'qubits': [2, 1]}]}],
 'header': {'backend_name': 'ibmq_manila', 'backend_version': '1.0.30'},
 'qobj_id': '3cd53092-1349-42d8-9b70-5bd8c76d2412',
 'schema_version': '1.3.0',
 'type': 'QASM'}
```

Qiskit and remote execution



Qiskit client

Qobj job

Job results

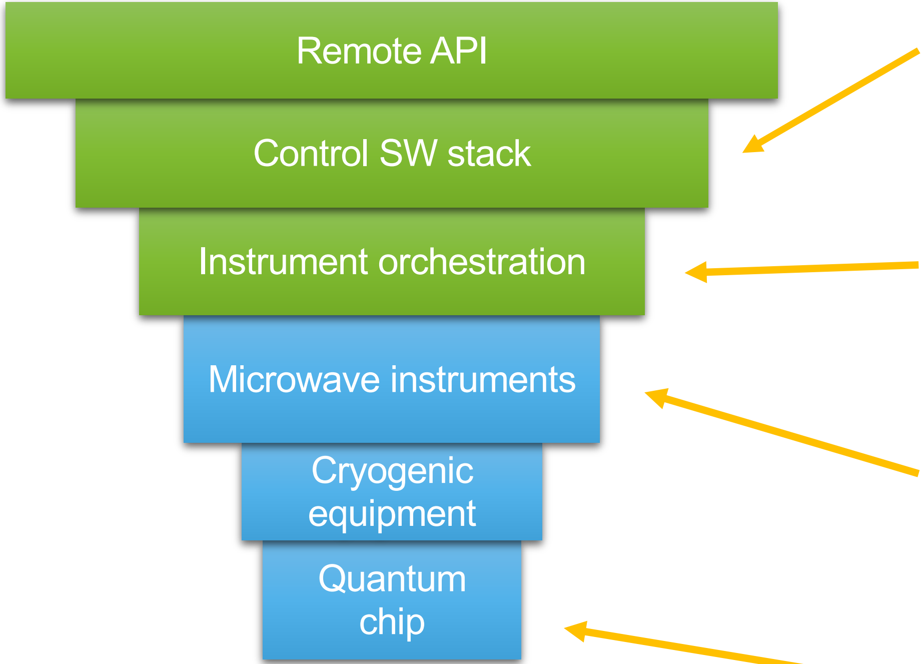
- User auth
- Configuration
- Properties
- Defaults
- Status

Backend’s REST API (with JSON serialization)

Quantum computer

Hardware level encoding

Quantum computer inner clock-work



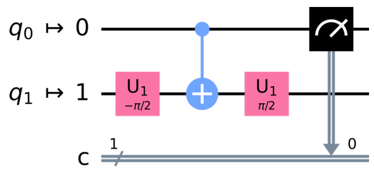
- Mapping from gates to pulses
- Routines for automatic calibration
- Internal database

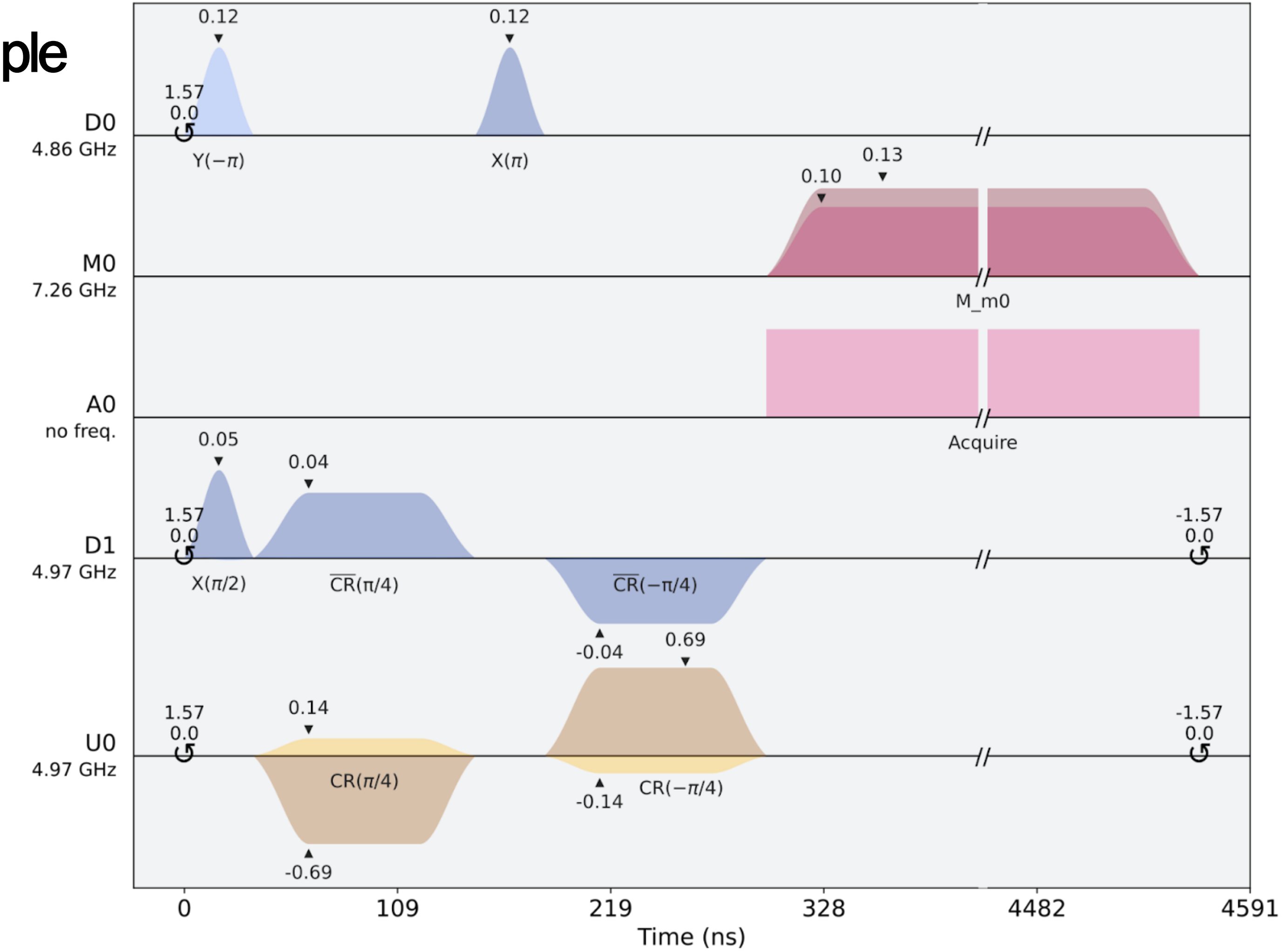
- Generate assembly instructions for digital signal processing (DSP)
- Instrument synchronization
- Data acquisition loop

- Instruments are pre-programmed
- There is no real-time feedback loop yet

- Quantum chip is just an electronic circuit
- We send a control mw-pulse and measure the corresponding response

Pulse schedule example



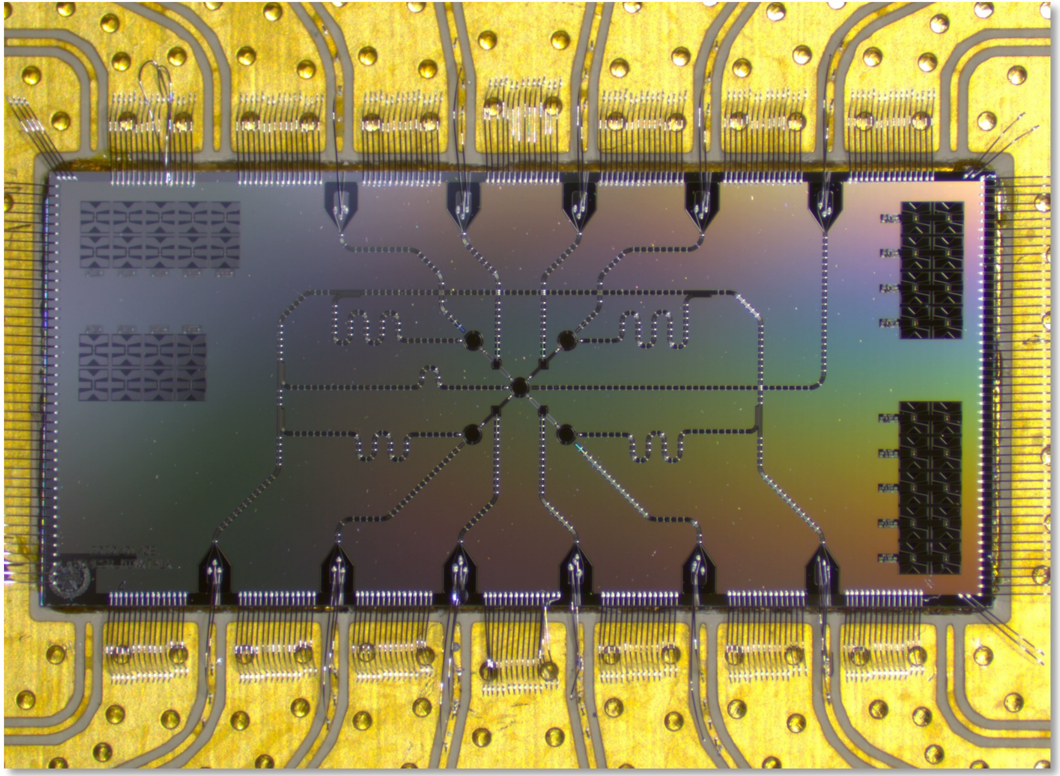


Qblox instruments assembly example

Q1ASM program:

```
        wait_sync      4
        upd_param      4
        set_mrk        15            # set markers to 15
        wait           4             # Latency correction of 0 ns.
        move           2000,R0       # iterator for loop with label start
    start:
        reset_ph
        upd_param      4
        wait           65532         # auto generated wait (300000 ns)
        wait           65532         # auto generated wait (300000 ns)
        wait           65532         # auto generated wait (300000 ns)
        wait           65532         # auto generated wait (300000 ns)
        wait           37872         # auto generated wait (300000 ns)
        set_awg_gain   851,0         # setting gain for gaussian-d1-0
        play           0,1,4         # play gaussian-d1-0 (100 ns)
        wait           96            # auto generated wait (96 ns)
        wait           4             # auto generated wait (4 ns)
        set_awg_gain   851,0         # setting gain for gaussian-d1-104
        play           0,1,4         # play gaussian-d1-104 (100 ns)
        wait           3596          # auto generated wait (3596 ns)
        loop           R0,@start
        set_mrk        0             # set markers to 0
        upd_param      4
        stop
```

Quantum chip



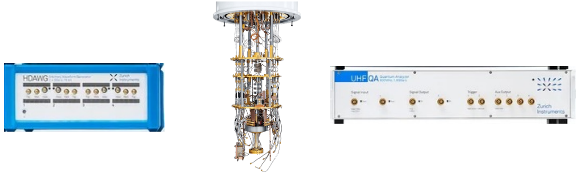
Device designed and measured by Christopher Warren.
Photo courtesy of Christopher Warren.

LUMI – QAL9000 PoC connection

QAL-9000

A dedicated experimental setup at Chalmers, work-in-progress

Gen1

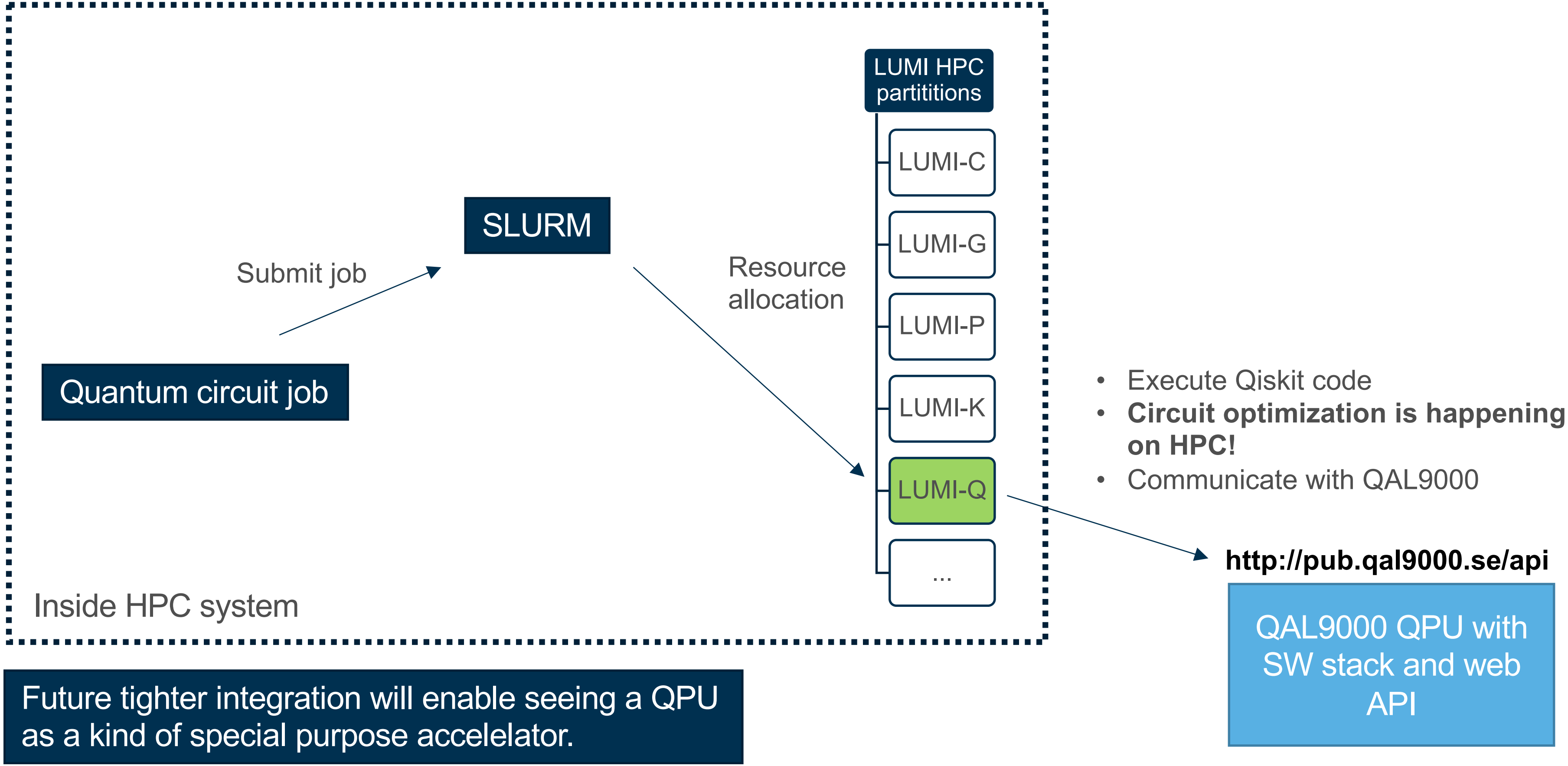


Labber, ZI, 5 qubits

Gen2

Quantify, Qblox, 7-20 qubits

LUMI ⟷ QAL9000



Quantum circuit job

Submit job

SLURM

Resource allocation

LUMI HPC partititions

- LUMI-C
- LUMI-G
- LUMI-P
- LUMI-K
- LUMI-Q
- ...

Inside HPC system

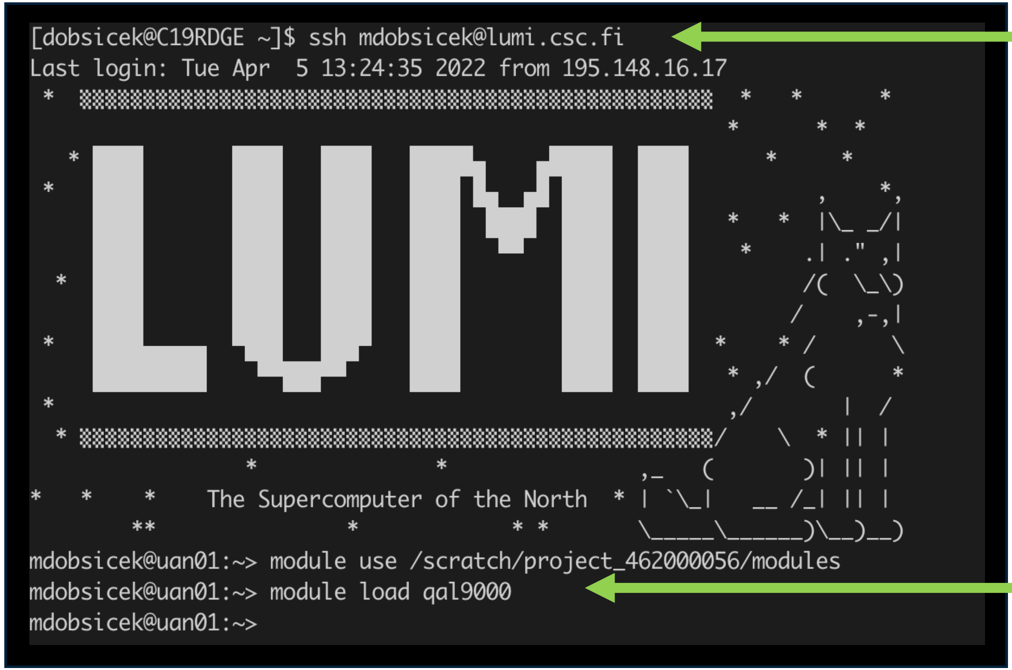
- Execute Qiskit code
- **Circuit optimization is happening on HPC!**
- Communicate with QAL9000

http://pub.qal9000.se/api

QAL9000 QPU with SW stack and web API

Future tighter integration will enable seeing a QPU as a kind of special purpose accelelator.

Prerequisites



```
[dobsicek@C19RDGE ~]$ ssh mdobsicek@lumi.csc.fi
Last login: Tue Apr  5 13:24:35 2022 from 195.148.16.17

The Supercomputer of the North

mdobsicek@uan01:~> module use /scratch/project_462000056/modules
mdobsicek@uan01:~> module load qal9000
mdobsicek@uan01:~>
```

Remote connection to HPC

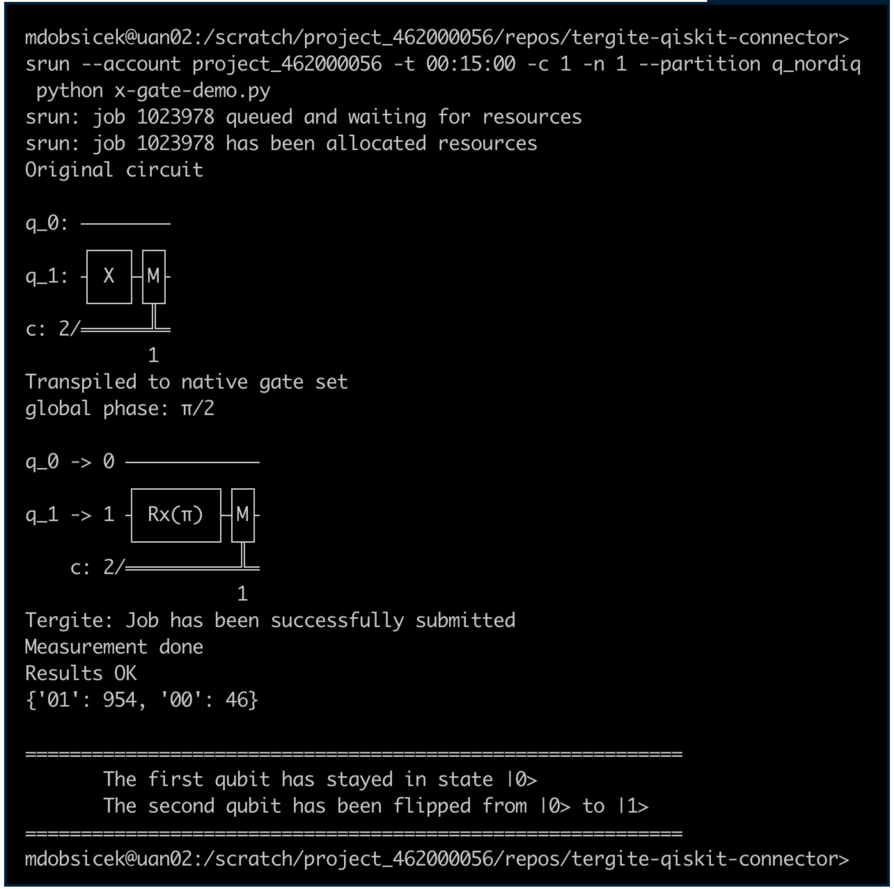
The NordIQuEst API will have a shape of an HPC module!

First test

The Qiskit program

```
#!/usr/bin/env python

from qiskit.providers.tergite import Tergite
import qiskit.compiler as compiler
import qiskit.circuit as circuit

provider = Tergite.get_provider()
backend = provider.get_backend("Pingu")
backend.set_options(shots=1000)

# Create circuit
qc = circuit.QuantumCircuit(2, 2)
qc.x(1)
qc.measure(1, 1)

# Transpile to native gate set
tc = compiler.transpile(qc, backend=backend)

# Compile to OpenPulse schedule
sched = compiler.schedule(tc, backend=backend)

# Submit the schedule for execution
job = backend.run(
    sched,
    meas_level=2,
    meas_return="single",
    qobj_header={
        "tag" : "Miroslav experiment",
    },
)
```

Execution



```
mdobsicek@uan02:/scratch/project_462000056/repos/tergite-qiskit-connector>
srun --account project_462000056 -t 00:15:00 -c 1 -n 1 --partition q_nordiq
 python x-gate-demo.py
srun: job 1023978 queued and waiting for resources
srun: job 1023978 has been allocated resources
Original circuit

q_0: ────────
     ┌───┐┌─┐
q_1: ┤ X ├┤M├
     └───┘└╥┘
c: 2/══════╩═
           1
Transpiled to native gate set
global phase: π/2

q_0 -> 0 ────────────
         ┌───────┐┌─┐
q_1 -> 1 ┤ Rx(π) ├┤M├
         └───────┘└╥┘
    c: 2/══════════╩═
                   1
Tergite: Job has been successfully submitted
Measurement done
Results OK
{'01': 954, '00': 46}

===========================================================
       The first qubit has stayed in state |0>
       The second qubit has been flipped from |0> to |1>
===========================================================
mdobsicek@uan02:/scratch/project_462000056/repos/tergite-qiskit-connector>
```

Thank you.