COE 530
Quantum Computers and Architecture

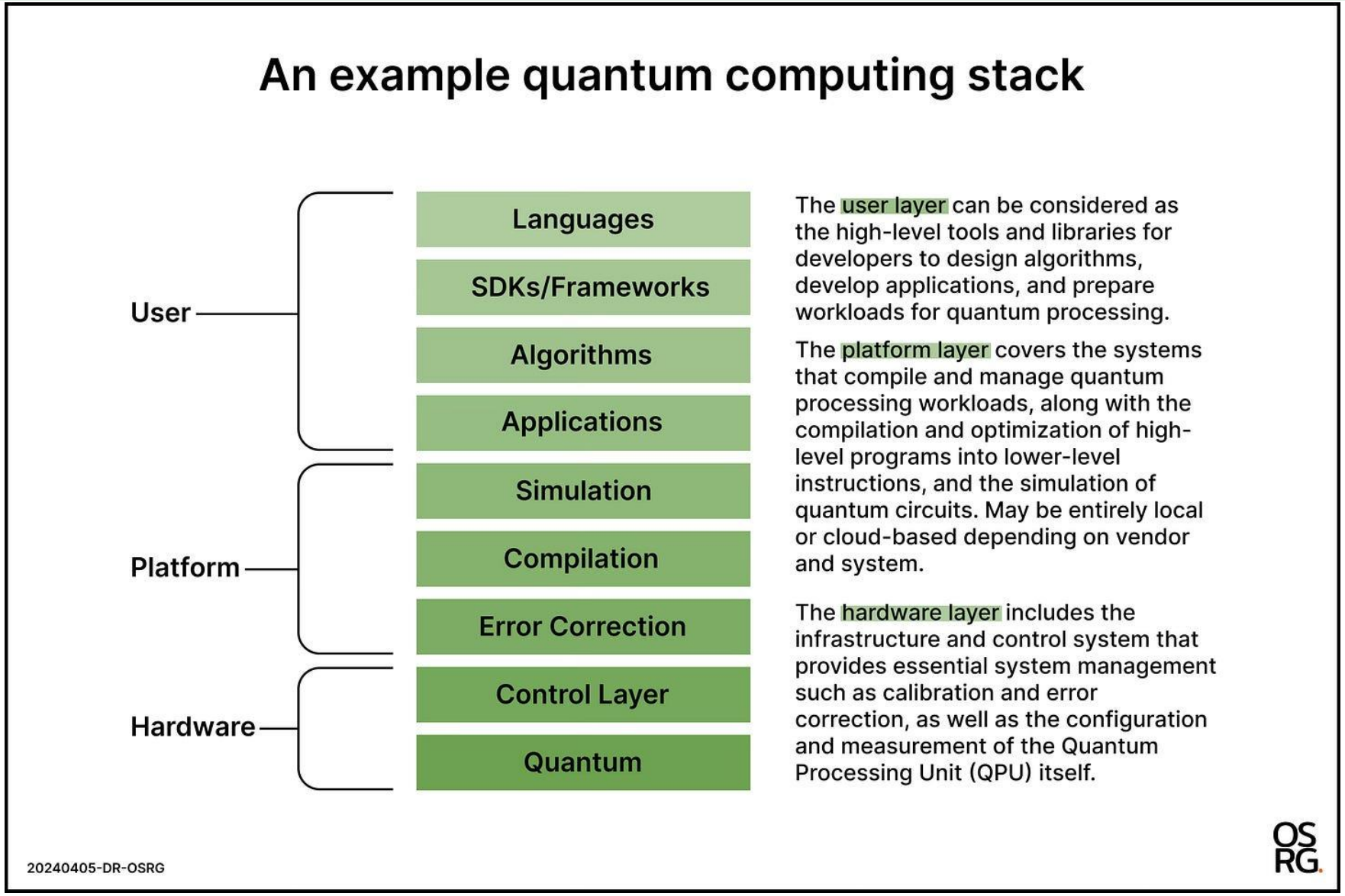
Lecture 8

Quantum Computing Systems – Quantum Assembly Language

References:

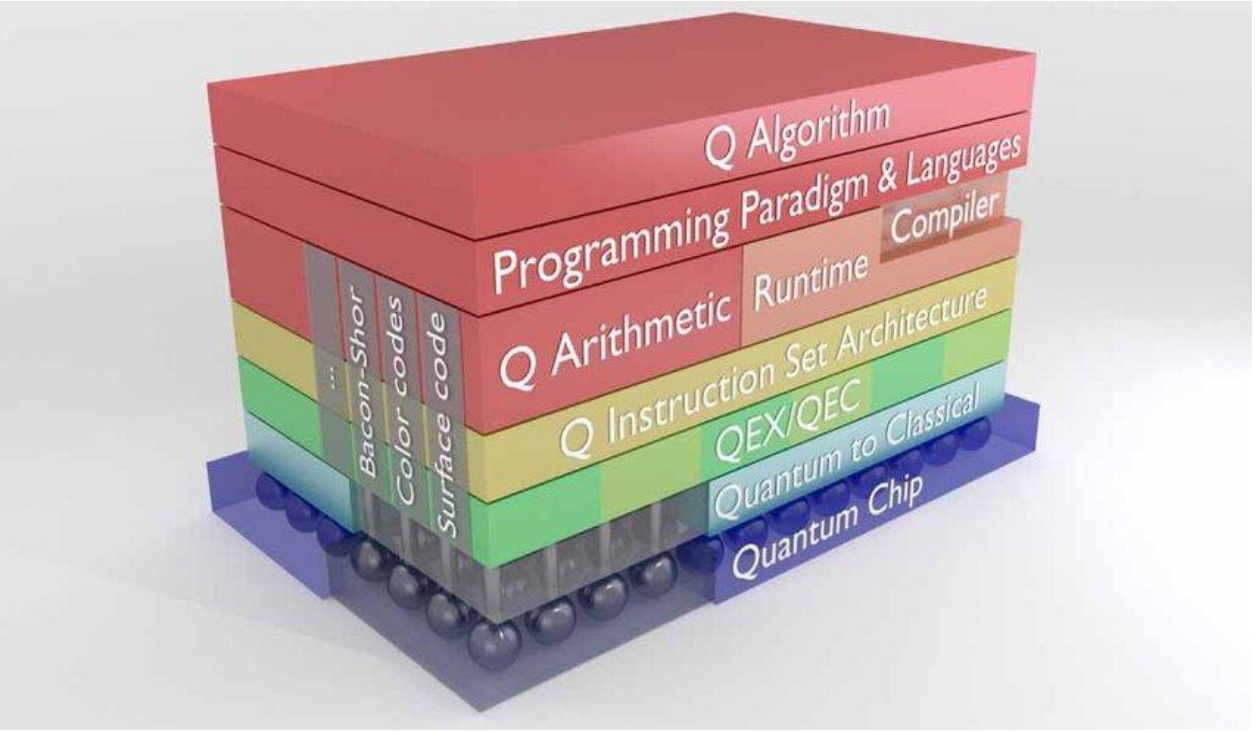
Ch 4,6 Quantum Computer Systems, Ding and Chong
Open Quantum Assembly Language
OpenQASM 3: A Broader and Deeper □antum Assembly Language

Quantum Computing Stack



https://productindeep.substack.com/p/whats-in-a-quantum-stack

Quantum Computing Stack



Charbon E, Sebastiano F, Vladimirescu A, Homulle H, Visser S, Song L and Incandela R M 2016 Cryo-cmos for quantum computing 2016 IEEE Int. Electron Devices Meeting (IEDM) pp

Quantum Computing Stack

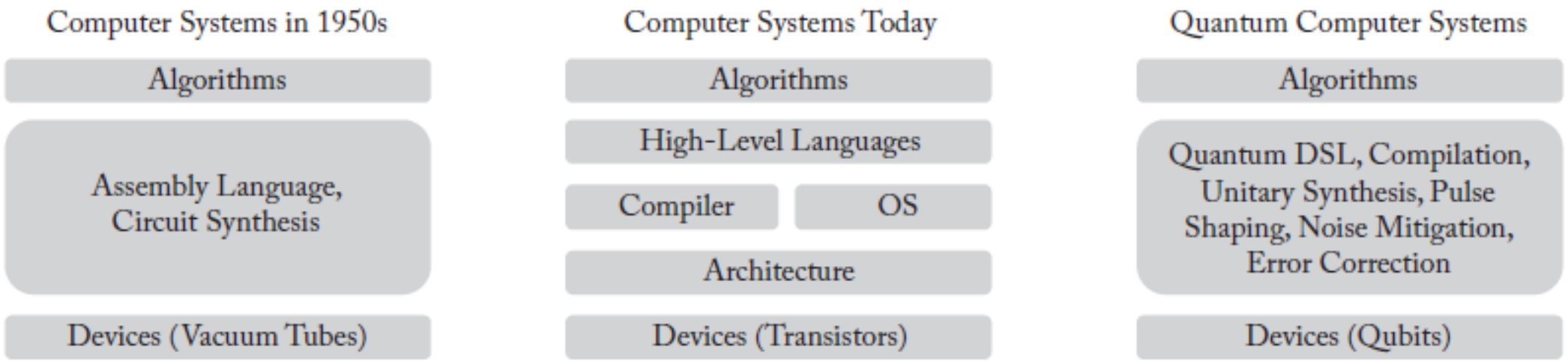


Figure 1.2: Architectural designs of classical vs. quantum computers. The abstraction layers for 1950s classical computing, today's classical computing, and quantum computing are compared.

Quantum Computer Systems: Research for Noisy Intermediate-Scale Quantum Computers

Layers of Quantum Computer

- The key to a successful design of a quantum computers is **abstraction**
- Most of today's QC systems consists of 3 layers
  - Application layer
  - Systems software layer
  - Hardware layer
- Each layer hides some implementations details for the next (above) layer

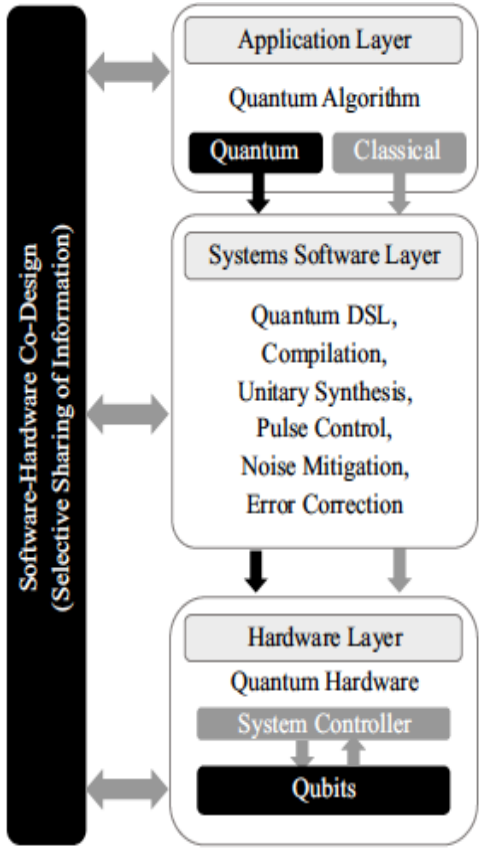
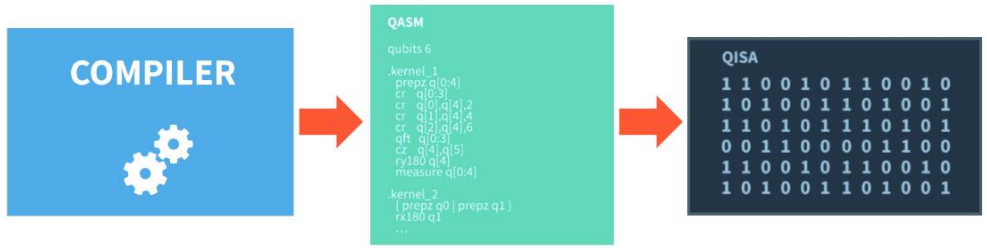


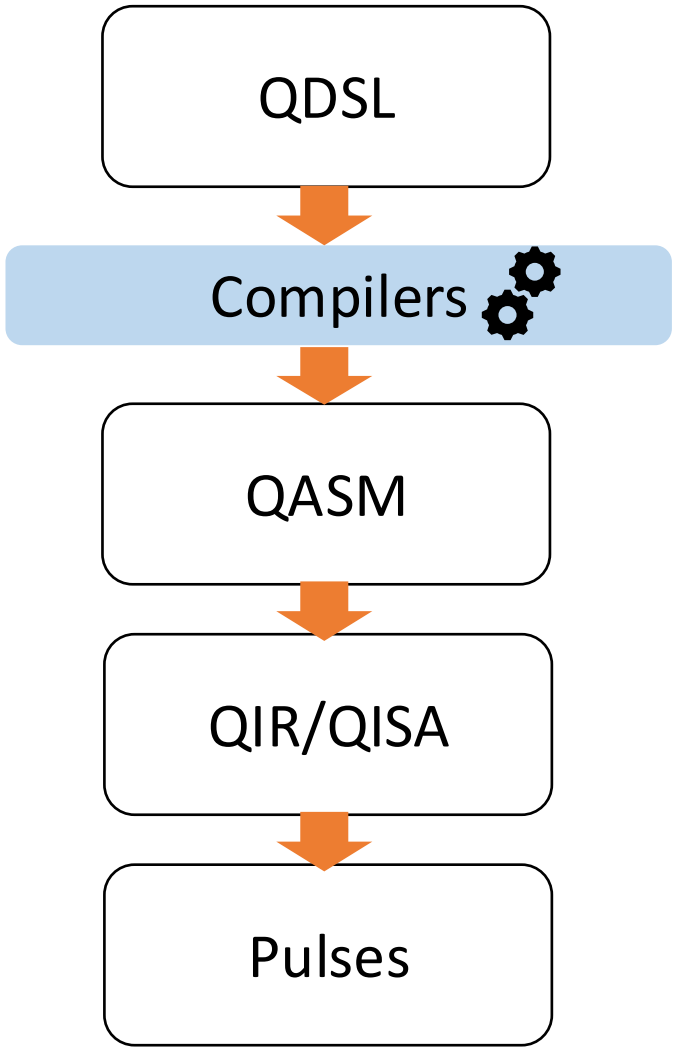
Figure 4.1: Selective sharing of information allows algorithms to use limited resource in NISQ hardware most efficiently.

Quantum Computer Systems: Research for Noisy Intermediate-Scale Quantum Computers

Quantum Compilers

- A quantum algorithm (or application) is implemented in a Quantum Domain-Specific Language (**QDSL**)
  - E.g., Qiskit, myQLM, CirQ, etc
- The quantum compiler translates the high-level program into Quantum Assembly Code (**QASM**) that can be executed on quantum hardware
  - E.g. OpenQASM
- Each line in QASM is translated to a series of Quantum Intermediate Representation (QIR/QISA)
- Each QIR instruction is translated into hardware control pulses





Difference Between QASM and QIR

QASM

Quantum Assembly Language

- Human-readable text format describing quantum circuits
- Specifies gates, qubits, and measurements at the circuit level
- Hardware-agnostic; portable across platforms (e.g., OpenQASM)
- Analogous to classical assembly language

Where It's Used

- Output of quantum compilers (Qiskit, Cirq)
- Circuit exchange between frameworks
- Quantum simulators and cloud APIs

QIR

Quantum Intermediate Representation

- Lower-level representation closer to machine instructions
- Enables compiler optimizations (e.g., gate fusion, routing)
- Hardware-specific; maps to device topology and native gate set
- Analogous to classical IR (e.g., LLVM IR)

Where It's Used

- Backend compilation to control pulses
- Hardware-specific optimization passes
- Qubit mapping and gate scheduling

Generic Quantum Compilation Flow

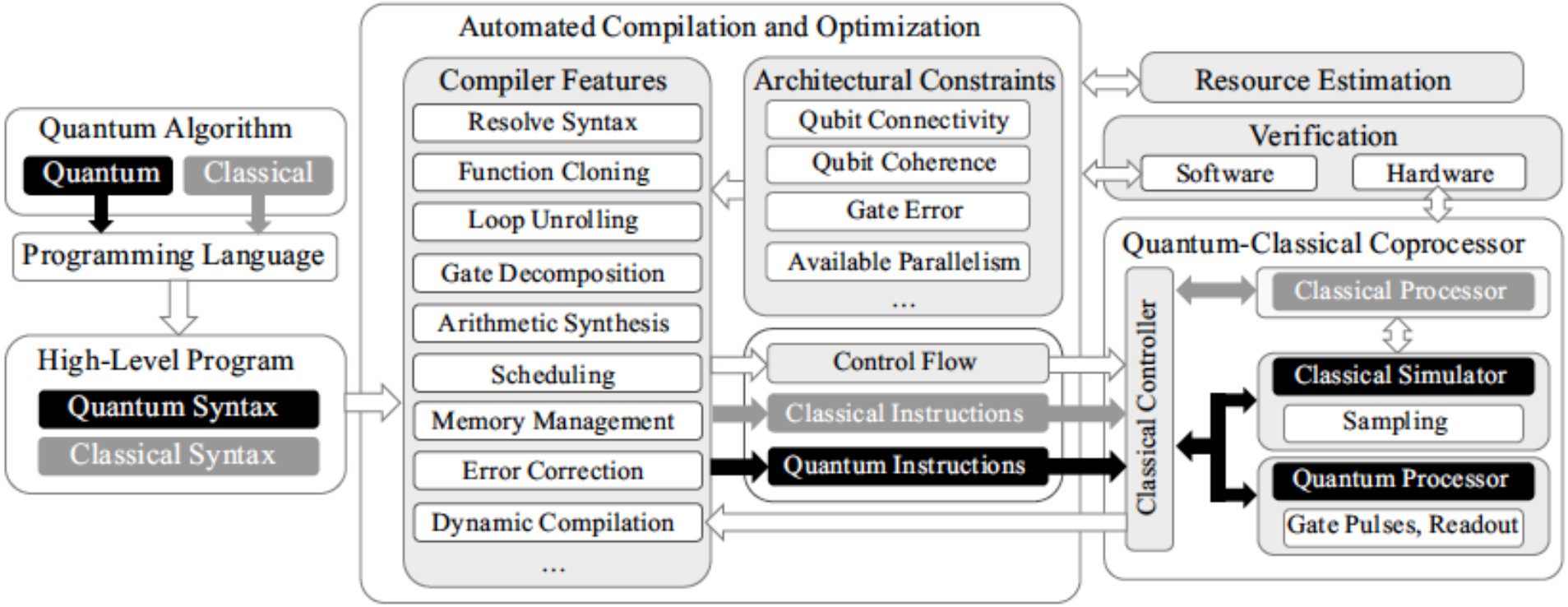
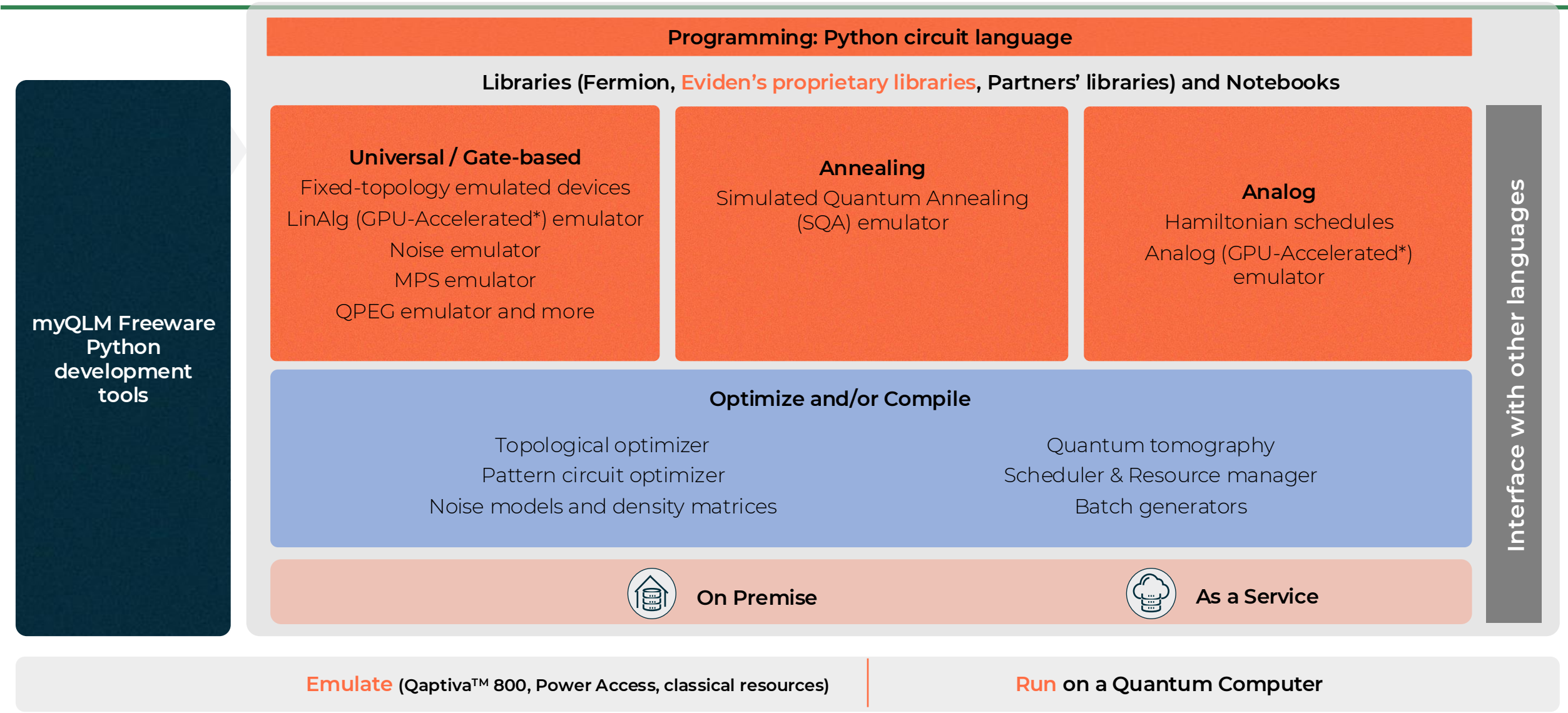


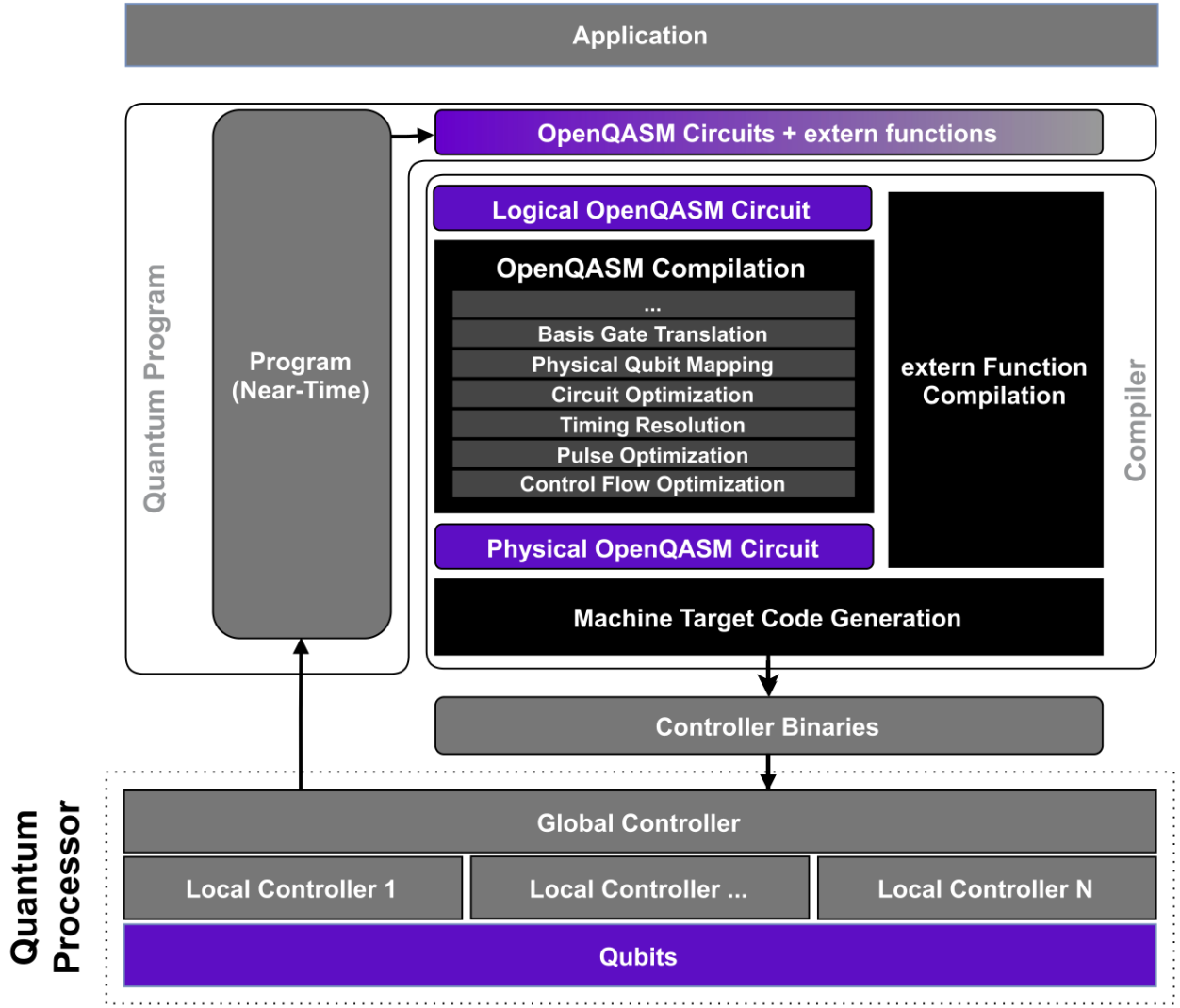
Figure 4.2: A detailed quantum compilation flow outlining the transformations and optimizations involved in a generic compiler.

Quantum Computer Systems: Research for Noisy Intermediate-Scale Quantum Computers

Use case: Qaptiva™ Application platform



Use case: OpenQASM3.0 compilation and execution model



OpenQASM 3: A Broader and Deeper □antum Assembly Language

Quantum Architecture Constraints

- Building a quantum computing systems must satisfy the following architectural constraints
  - **Instruction set:** high-level programs are translated into set of instructions sets defined by the universal set of gates (e.g. Clifford + T gates)
  - **Quantum communication:** practical multiple qubit systems require moving qubits closer to each other via chains of swap gates
  - **Hardware noise:** minimize errors caused by hardware noise
    - Memory errors -> decoherence of qubits
    - Gate errors -> imprecise control of gates
  - **Available parallel control:** parallelism is limited by the hardware control mechanisms or error mitigation protocols.

What is a qubit? DiVincenzo Criteria

- In 2000, David DiVincenzo proposed five criteria that any physical system must satisfy to serve as a viable qubit for quantum computation:
  1. **A scalable physical system with well-characterized qubits**
  2. **Ability to initialize the state of the qubits** to a simple fiducial state (e.g., $|000...0\rangle$)
  3. **Long relevant decoherence times**, much longer than the gate operation time (T_1 and T_2)
  4. **A universal set of quantum gates** (single-qubit + two-qubit gates such as CNOT)
  5. **A qubit-specific measurement capability** to read out the result

- Plus two additional criteria for quantum communication:
  1. **Ability to interconvert** stationary and flying qubits
  2. **Ability to faithfully transmit** flying qubits between specified locations

Objective of OpenQASM

What is OpenQASM?

- An imperative programming language for describing quantum circuits
- Serves as an intermediate representation (IR) between high-level compilers and quantum hardware
- Machine-independent — not tied to any specific vendor or QPU

OpenQASM is a quantum circuit intermediate representation (IR):

- Based on the **quantum circuit model** — a standard formalism for quantum computation
- Human-readable text format (pronounced "kazm")
- Represents **universal physical circuits** with a parameterized gate set

Execution Model (4 Phases):

1. **Compilation** — offline, classical computer, problem-independent
2. **Circuit generation** — online, with problem parameters known
3. **Execution** — real-time on quantum computer controllers
4. **Post-processing** — classical analysis of measurement results

Syntax: Registers and Built-in Gates

Header and Registers:

- `OPENQASM 2.0;` — version declaration (must be first line)
- `qreg q[5];` — quantum register of 5 qubits (initialized to $|0\rangle$)
- `creg c[5];` — classical register of 5 bits (initialized to 0)

Built-in Universal Gate Basis — "CNOT + U(2)":

- `U(θ,ϕ,λ) q[0];` — arbitrary single-qubit gate: $U = Rz(\phi)\cdot Ry(\theta)\cdot Rz(\lambda)$
- `CX q[0],q[1];` — controlled-NOT (CNOT) entangling gate
- Example: `U(pi/2,0,pi) q[0];` applies a Hadamard gate

Other Statements:

- `measure q -> c;` — Z-basis measurement, records result in classical bits
- `reset q[0];` — prepare qubit in state $|0\rangle$
- `barrier q;` — prevent optimizer from reordering across this line

Syntax: Gate Subroutines and Classical Control

Custom Gate Subroutines:

New unitary gates are defined hierarchically from built-in gates:

```
gate h a { U(pi/2,0,pi) a; }
gate cu1(lambda) a,b {
  U(0,0,lambda/2) a; CX a,b;
  U(0,0,-lambda/2) b; CX a,b;
  U(0,0,lambda/2) b; }
```

- Gates must be declared before use and cannot call themselves
- Standard header `qelib1.inc` provides: x, y, z, h, s, t, cx, ccx, cu1, etc.

Classical Control — the if Statement:

```
if(c==1) z q[2];  // apply Z only if c equals 1
```

- Conditionally execute a quantum op based on a classical register value
- Register interpreted as integer (bit[0] = low-order bit)
- Enables measurement outcomes to determine future quantum operations

OpenQASM Syntax

Table 1: Open QASM language statements (version 2.0)

Statement	Description	Example
`OPENQASM 2.0;`	Denotes a file in Open QASM format[a]	`OPENQASM 2.0;`
`qreg name[size];`	Declare a named register of qubits	`qreg q[5];`
`creg name[size];`	Declare a named register of bits	`creg c[5];`
`include "filename";`	Open and parse another source file	`include "qelib1.inc";`
`gate name(params) qargs { body }`	Declare a unitary gate	(see text)
`opaque name(params) qargs;`	Declare an opaque gate	(see text)
`// comment text`	Comment a line of text	`// oops!`
`U(theta,phi,lambda) qubit|qreg;`	Apply built-in single qubit gate(s)[b]	`U(pi/2,2*pi/3,0) q[0];`
`CX qubit|qreg,qubit|qreg;`	Apply built-in CNOT gate(s)	`CX q[0],q[1];`
`measure qubit|qreg -> bit|creg;`	Make measurement(s) in Z basis	`measure q -> c;`
`reset qubit|qreg;`	Prepare qubit(s) in $\|0\rangle$	`reset q[0];`
`gatename(params) qargs;`	Apply a user-defined unitary gate	`crz(pi/2) q[1],q[0];`
`if(creg==int) qop;`	Conditionally apply quantum operation	`if(c==5) CX q[0],q[1];`
`barrier qargs;`	Prevent transformations across this source line	`barrier q[0],q[1];`

[a] This must appear as the first non-comment line of the file.

[b] The parameters `theta`, `phi`, and `lambda` are given by *parameter expressions*; see text and Appendix A.

Example: Quantum Ripple-Carry Adder

Adds a = 0001 to b = 1111 using Cuccaro et al. (quant-ph/0410184):

```
OPENQASM 2.0; include "qelib1.inc";
gate majority a,b,c
  { cx c,b; cx c,a; ccx a,b,c; }
gate unmaj a,b,c
  { ccx a,b,c; cx c,a; cx a,b; }
qreg cin[1]; qreg a[4]; qreg b[4];
qreg cout[1]; creg ans[5];
x a[0]; x b;  // a=0001, b=1111
majority cin[0],b[0],a[0];
majority a[0],b[1],a[1];
majority a[1],b[2],a[2];
majority a[2],b[3],a[3];
cx a[3],cout[0];
unmaj a[2],b[3],a[3]; ... unmaj cin[0],b[0],a[0];
measure b[0] -> ans[0]; ... measure cout[0] -> ans[4];
```

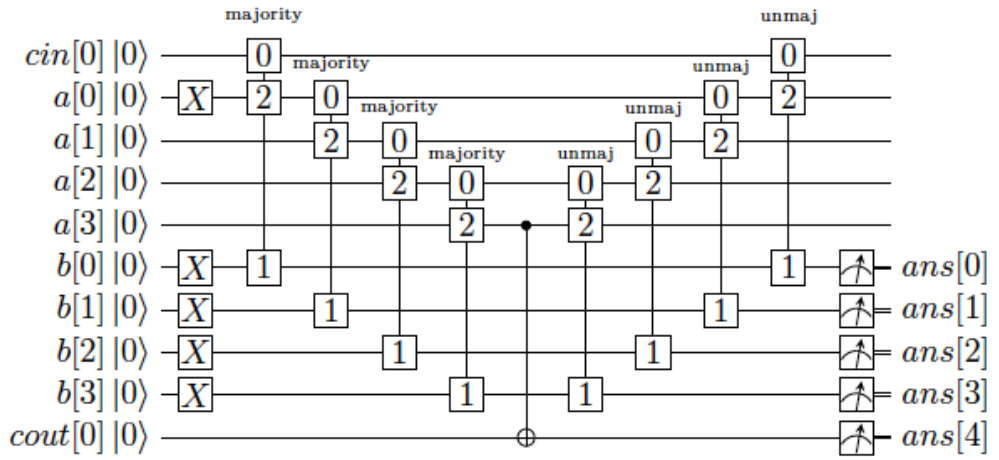
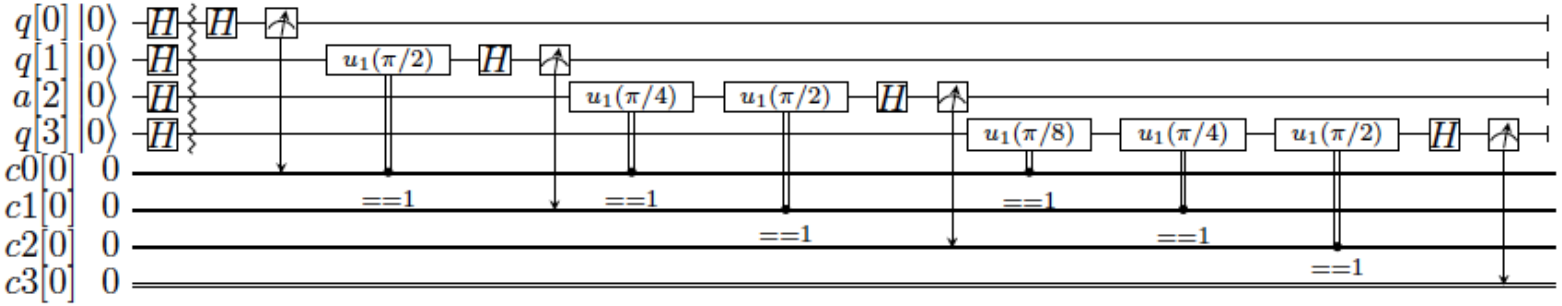


Figure 11: Example of a quantum ripple-carry adder from [29]. This circuit prepares $a = 1$, $b = 15$ and computes the sum into `b` with an output carry `cout[0]`.

Example: Quantum Fourier Transform

Applies QFT to $|q_0q_1q_2q_3\rangle = |1010\rangle$ and measures in computational basis:

```
OPENQASM 2.0; include "qelib1.inc";
qreg q[4]; creg c[4];
x q[0]; x q[2];  // prepare |1010⟩
barrier q;
h q[0];
cu1(pi/2) q[1],q[0]; h q[1];
cu1(pi/4) q[2],q[0]; cu1(pi/2) q[2],q[1]; h q[2];
cu1(pi/8) q[3],q[0]; cu1(pi/4) q[3],q[1];
cu1(pi/2) q[3],q[2]; h q[3];
measure q -> c;
```



Key points:

- Demonstrates **parameter passing** to gate subroutine cu1(λ)
- Each qubit gets H + controlled phase rotations ($\pi/2^k$)
- Output read in reverse order: c[3], c[2], c[1], c[0]

Motivation: Why OpenQASM 3?

OpenQASM 2 Limitations:

- Limited to qubits and gates — no classical control flow
- No timing control or pulse-level descriptions
- Cannot express modern experiments (e.g., repeat-until-success)

New Needs Driving OpenQASM 3:

- Repeat-until-success algorithms require classical branching
- Calibration and error mitigation need timing control
- Dynamical decoupling demands precise pulse scheduling
- Multi-level IR needed for circuit development and optimization

OpenQASM 3: Syntax and Structure

Program Structure:

- Header: OPENQASM 3; include "stdgates.inc";
- Declarations: qubit[n] q; bit[n] c; int, uint, float, bool
- Operations: gates, measure, reset, barrier

Gates:

- Built-in: U(θ, ϕ, λ) and gphase(γ)
- Custom: gate h a { U(π/2, 0, π) a; }
- Modifiers: ctrl @, inv @, pow(r) @

New in Version 3:

- Control flow: if/else, for, while loops
- Subroutines: def, extern (external classical functions)
- Timing: duration, stretch, delay, box
- Pulse definitions: defcal for gate calibration

Two Timescales of Quantum-Classical Interaction

Real-Time Classical Computation:

- Must execute within qubit coherence times
- Tightly coupled to quantum operations on the QPU
- Examples: mid-circuit measurement, feed-forward, conditional ops
- **This is the focus of OpenQASM 3**

Near-Time Classical Computation:

- Less stringent timing requirements
- Pre/post-processing (e.g., VQE outer loop, Shor co-prime)
- Handled by existing classical programming frameworks

Classical Computation Extensions

Classical Types:

- int, uint, float, bool, bit — familiar C-like types
- Parameterized widths: int[8], float[64], bit[n]

Control Flow:

- if / else branching based on measurement outcomes
- for and while loops for iterative algorithms
- Subroutines (def) for modular circuit definitions

External Functions (extern):

- Opaque references to classical computations on run-time data
- Enables reuse of existing classical tooling

Gate Modifiers

OpenQASM 3 introduces **gate modifiers** to build new gates from existing ones:

ctrl @ gate — adds a control qubit to any gate

- e.g., ctrl @ x = CNOT; ctrl @ ctrl @ x = Toffoli

inv @ gate — computes the inverse (adjoint) of a gate

- Compiler reverses gate sequence and inverts each gate

pow(r) @ gate — computes the r-th power of a gate

- e.g., pow(0.5) @ z = S gate (square root of Z)

Modifiers can be **composed**: ctrl @ inv @ pow(2) @ s

Timing and Pulse Control

Explicit Timing:

- **duration** type — specify delays in ns, us, or backend dt units
- **stretch** type — compiler-resolvable flexible duration
- **delay** instruction — insert idle time on specific qubits
- **box** statement — scope timing constraints for optimization

Embedded Pulse-Level Definitions (defcal):

- Attach low-level pulse implementations to gates
- Enables calibration: optimize pulse parameters via gate-level circuits
- Uses extern grammar (e.g., OpenPulse) for waveform descriptions

Multi-Level IR and Compilation Model

OpenQASM 3 serves as an IR at **multiple levels of abstraction**:

1. **High-Level (Logical)** — virtual qubits, gate modifiers, subroutines
2. **Mid-Level (Circuit)** — basis gates, qubit mapping, timing constraints
3. **Low-Level (Physical)** — pulse definitions, physical qubits ($0, $1)

Compilation Flow:

- Application emits quantum payloads as OpenQASM circuits
- Compiler transforms: basis gates, qubit mapping, timing, pulses
- Same language used from development through calibration

Use Cases and Applications

Dynamic Circuits:

- Mid-circuit measurements with conditional operations
- Repeat-until-success (e.g., Rz synthesis from Nielsen & Chuang)
- Iterative Phase Estimation with just one extra qubit

Dynamical Decoupling:

- Timing-sensitive pulse sequences expressed with stretch
- Compiler resolves to exact durations at compile time

Calibration and Noise Mitigation Examples

Calibration (defcal — Pulse-Level Gate Definitions):

- defcal attaches low-level pulse implementations to gates
- Optimize pulse parameters via error amplification sequences
- **Example:** Choose a gate with shorter duration but less-accurate calibration for a variational ansatz — minimizing decoherence matters more than exact unitary

Noise Mitigation (Dynamical Decoupling via Timing):

- Use stretch and delay to express timing-sensitive pulse sequences
- Captures design intent independent of exact gate durations
- **Example:** Spin-echo sequences — choose specific delay ratios during idle periods to cancel errors from slowly drifting hardware parameters

Key Insight: OpenQASM 3 enables the same language to be used from algorithm development through calibration and error mitigation — no separate tools needed

Example: Quantum Ripple-Carry Adder

4-bit adder (Cuccaro et al.) — adds a = 0001 to b = 1111:

```
OPENQASM 3; include "stdgates.inc";
gate majority a, b, c {
  cx c,b; cx c,a; ccx a,b,c; }
gate unmaj a, b, c {
  ccx a,b,c; cx c,a; cx a,b; }
qubit cin; qubit[4] a; qubit[4] b; qubit cout;
uint[4] a_in = 1; uint[4] b_in = 15;
for i in [0:3] {              // set input states
  if(bool(a_in[i])) x a[i];
  if(bool(b_in[i])) x b[i]; }
majority cin, b[0], a[0];
for i in [0:2] { majority a[i],b[i+1],a[i+1]; }
cx a[3], cout;
for i in [2:-1:0] { unmaj a[i],b[i+1],a[i+1]; }
measure b -> ans[0:3]; measure cout -> ans[4];
```

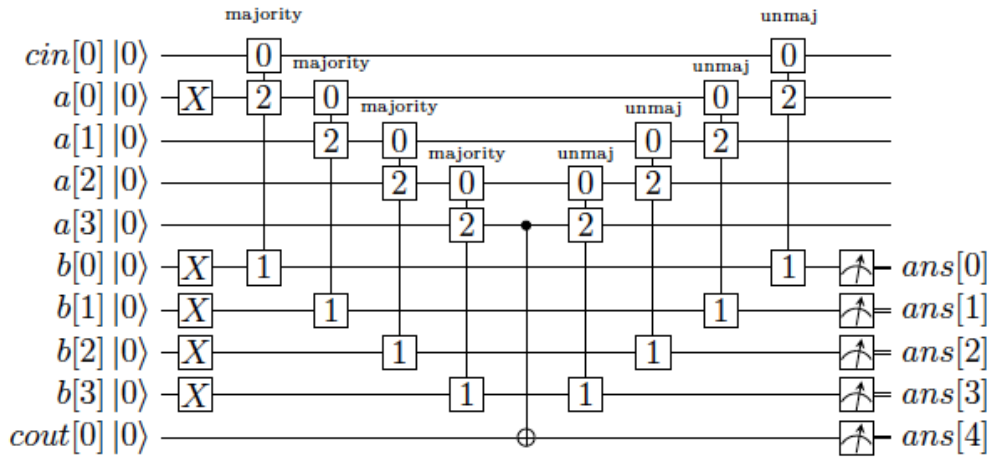


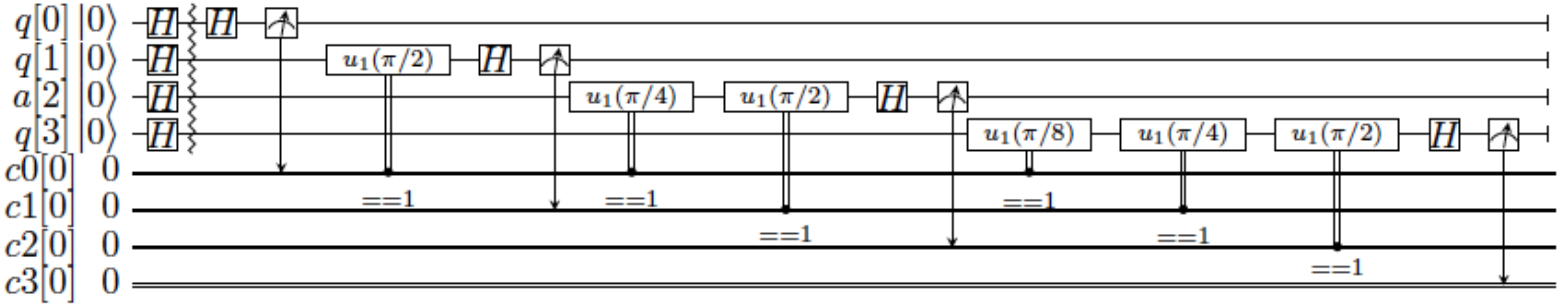
Figure 11: Example of a quantum ripple-carry adder from [29]. This circuit prepares $a = 1$, $b = 15$ and computes the sum into `b` with an output carry `cout[0]`.

Shows: **gate definitions**, **for loops**, **if statements**, **classical types** (uint), and **measurement**

Example: Quantum Fourier Transform (QFT)

Inverse QFT rewritten in OpenQASM 3 using loops and classical types:

```
OPENQASM 3; include "stdgates.inc";
qubit[4] q;  bit[4] c;  angle[4] theta = 0;
h q[0]; measure q[0] -> c[0];
for i in [1:3] {
  theta = 0;
  for j in [0:i-1] {
    theta[i-j] = c[j]; // bit-level ops
  }
  rz(theta) q[i]; h q[i];
  measure q[i] -> c[i];
}
```



Key advantages over OpenQASM 2:

- Nested for loops preserve iterative structure (vs. unrolled circuit)
- angle[4] type with bit-level ops converts measurements into rotations
- Mid-circuit measurement enables real-time classical feed-forward

How is OpenQASM different than AQASM?

- **OpenQASM**
  - An open, text-based quantum assembly language.
  - Developed originally by IBM
  - Used widely in Qiskit ecosystem
  - Current major versions: OpenQASM 2.0 and 3.0
- **Purpose**
  - Describe quantum circuits
  - Send circuits to quantum hardware or simulators
  - Interoperability between tools
- **Characteristics**
  - Human-readable
  - Hardware-oriented
  - Gate-level description
  - OpenQASM 3 adds:
  - Classical control flow
  - Timing
  - More hybrid quantum-classical features

- **AQASM**
  - Part of the **myQLM (Atos Quantum Learning Machine)** framework
  - Used to represent circuits in the QLM ecosystem
- **Purpose**
  - Internal circuit representation
  - Used for simulation, compilation, and execution within myQLM
  - Integrates tightly with Python APIs
- **Characteristics**
  - Not really meant as a universal interchange format
  - Designed for myQLM toolchain
  - Closely integrated with:
  - QLM simulators
  - myQLM compilers
  - Python-based workflow