Quantum Software Engineering MeetUp 2025

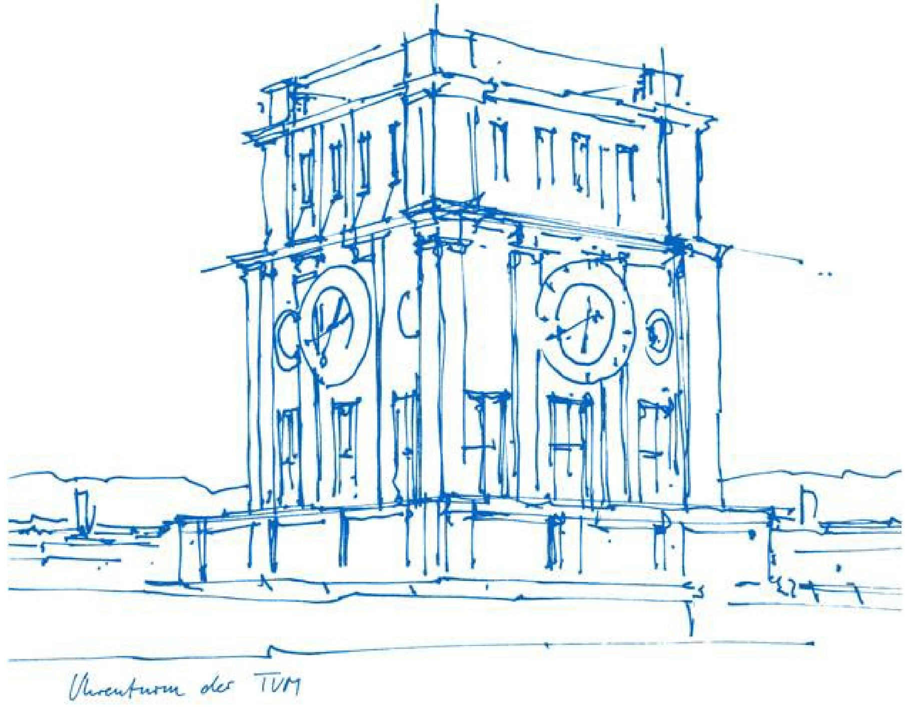
Bridge the Gap Between HPC Systems and Various Quantum Platforms: A Unified Quantum Platform

Amr Elsharkawy, **Xiaorang Guo**, Martin Schulz

Chair of Computer Architecture and Parallel Systems

Technical University of Munich (TUM)

February 24, 2025



Outline

- Background & Motivation
- Unified Quantum Platform
- Novel Quantum Control Processor
- Evaluation
- Conclusion & Future Work

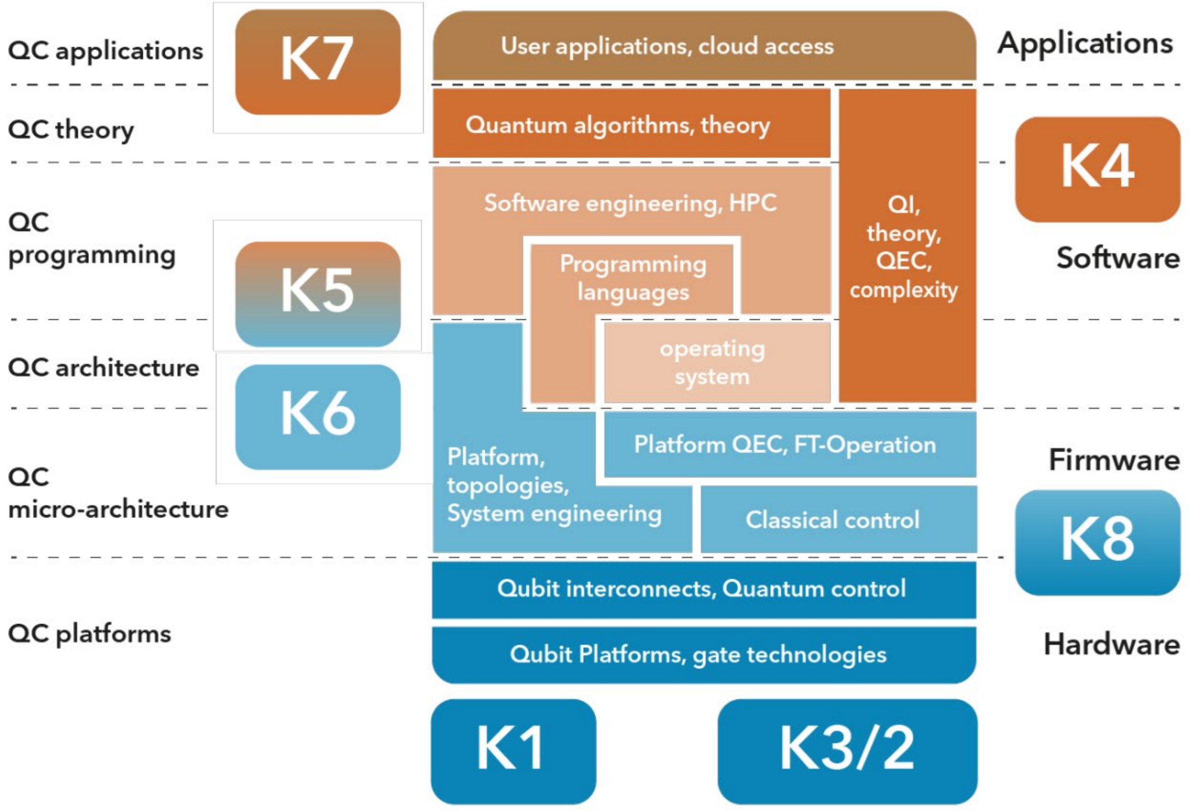
The Munich Quantum Valley: Full-Stack QC Systems

Three technologies; One stack

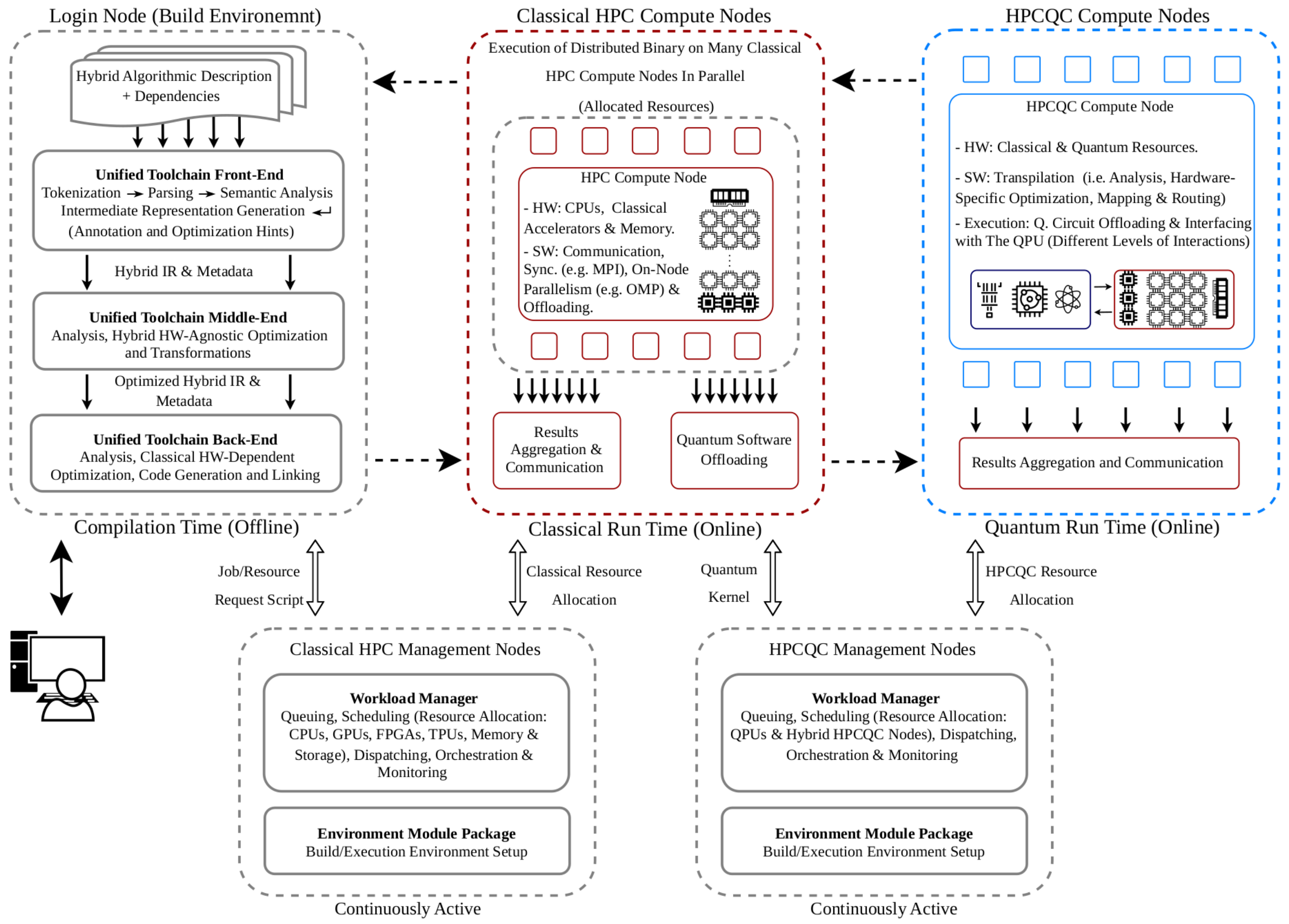
- Superconducting Qubits
- Neutral-atom Qubits
- Trapped-ion Qubits

Seven Consortia/Coordinated Projects

- Seven core partners
- Several associated projects with different funding organizations (Germany, EU, …)
- Plus Industrial Partners, Startups, and Lighthouse Projects
- Educational components


QC applications
QC theory
QC programming
QC architecture
QC micro-architecture
QC platforms
K7
K5
K6
User applications, cloud access
Quantum algorithms, theory
Software engineering, HPC
Programming languages
operating system
QI, theory, QEC, complexity
Platform, topologies, System engineering
Platform QEC, FT-Operation
Classical control
Qubit interconnects, Quantum control
Qubit Platforms, gate technologies
K1
K3/2
Applications
K4
Software
Firmware
K8
Hardware

The HPCQC Workflow


Login Node (Build Environemnt)
Hybrid Algorithmic Description + Dependencies
Unified Toolchain Front-End
Tokenization → Parsing → Semantic Analysis
Intermediate Representation Generation ↵
(Annotation and Optimization Hints)
Hybrid IR & Metadata
Unified Toolchain Middle-End
Analysis, Hybrid HW-Agnostic Optimization and Transformations
Optimized Hybrid IR & Metadata
Unified Toolchain Back-End
Analysis, Classical HW-Dependent Optimization, Code Generation and Linking
Compilation Time (Offline)
Classical HPC Compute Nodes
Execution of Distributed Binary on Many Classical HPC Compute Nodes In Parallel
(Allocated Resources)
HPC Compute Node
- HW: CPUs, Classical Accelerators & Memory.
- SW: Communication, Sync. (e.g. MPI), On-Node Parallelism (e.g. OMP) & Offloading.
Results Aggregation & Communication
Quantum Software Offloading
Classical Run Time (Online)
HPCQC Compute Nodes
HPCQC Compute Node
- HW: Classical & Quantum Resources.
- SW: Transpilation (i.e. Analysis, Hardware-Specific Optimization, Mapping & Routing)
- Execution: Q. Circuit Offloading & Interfacing with The QPU (Different Levels of Interactions)
Results Aggregation and Communication
Quantum Run Time (Online)
Job/Resource Request Script
Classical Resource Allocation
Quantum Kernel
HPCQC Resource Allocation
Classical HPC Management Nodes
Workload Manager
Queuing, Scheduling (Resource Allocation: CPUs, GPUs, FPGAs, TPUs, Memory & Storage), Dispatching, Orchestration & Monitoring
Environment Module Package
Build/Execution Environment Setup
Continuously Active
HPCQC Management Nodes
Workload Manager
Queuing, Scheduling (Resource Allocation: QPUs & Hybrid HPCQC Nodes), Dispatching, Orchestration & Monitoring
Environment Module Package
Build/Execution Environment Setup
Continuously Active

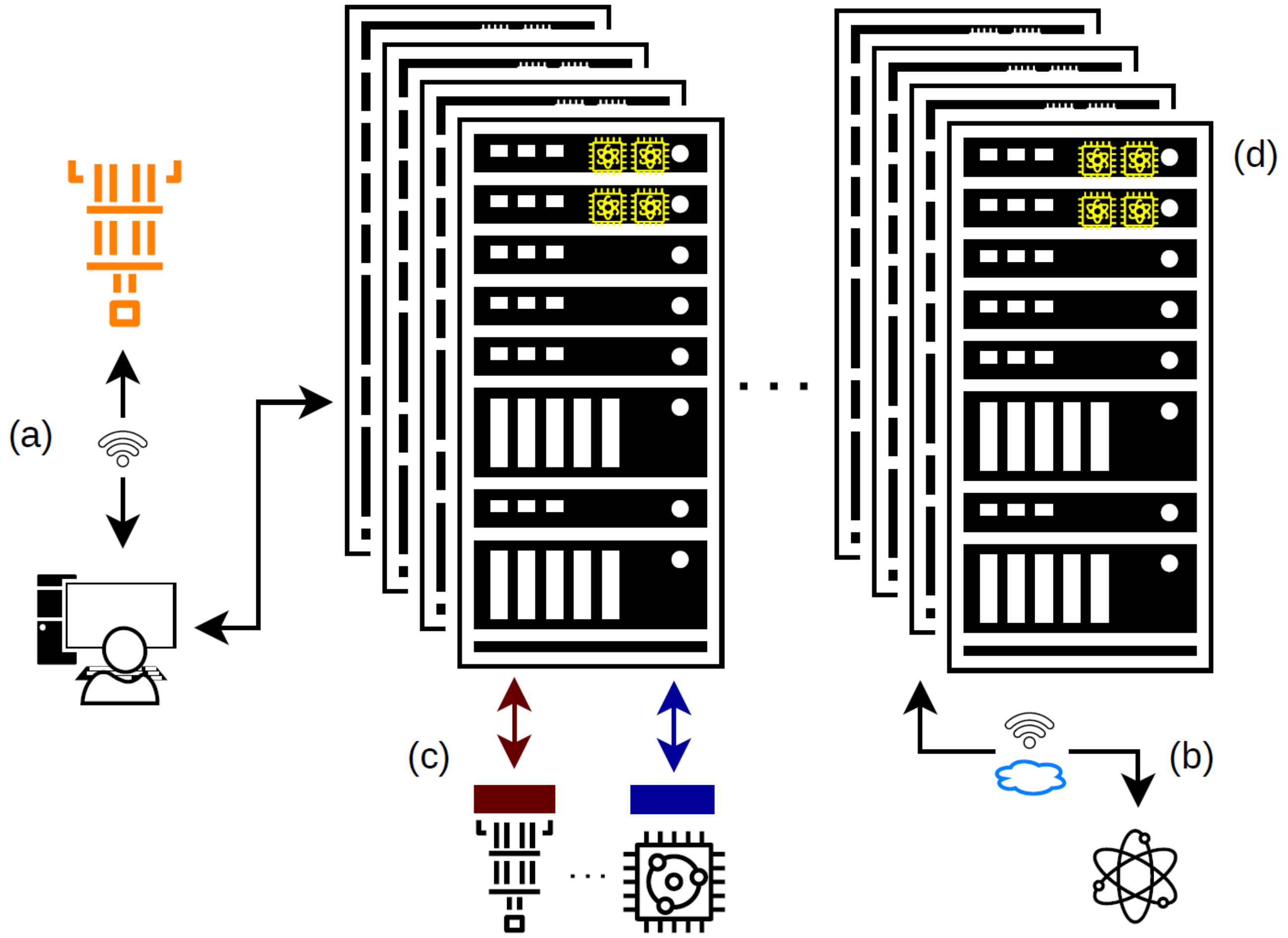
HPCQC Integration

SW-Level View

- Design of programming models
- Execution Schedulers
- Runtime Environments
- Seamless Integration

HW-Level View

(a) Loose Integration – Standalone
(b) Loose Integration – Co-located
(c) Tight Integration – Co-located
(d) Tight Integration – On-node.



Challenges on the Optimal Way

Cross-Technology Control

- Each existing backends only supports specific physical modality
- Demand an individual compiler for each backend controller

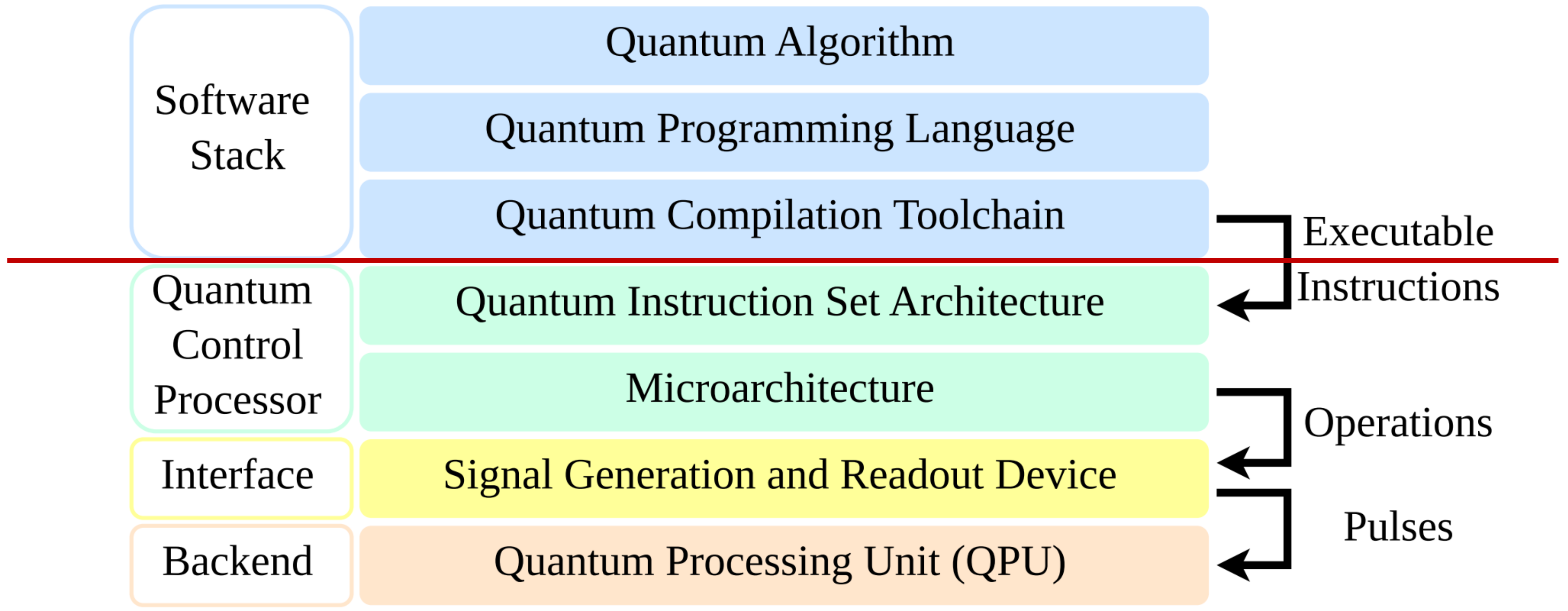
Communication Overhead for Conditional Logics

- Hybrid algorithms need large amount of communications between quantum and classical processors
  - Mid-circuit measurement & Feedforward logic
- Eat away quantum advantages
- Worse case: Exceeds coherence time

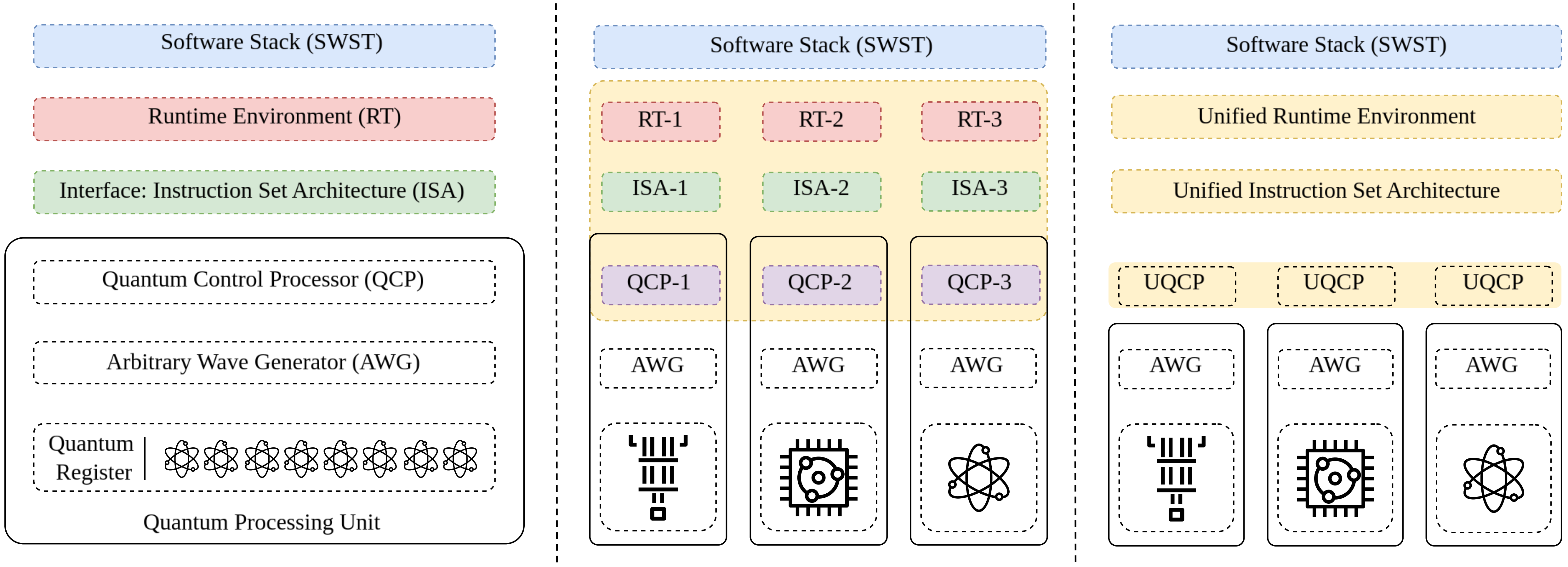
NEED!

Unified platform (Software environment & Control Processor)
to dampen the overhead and achieve cross-technology control

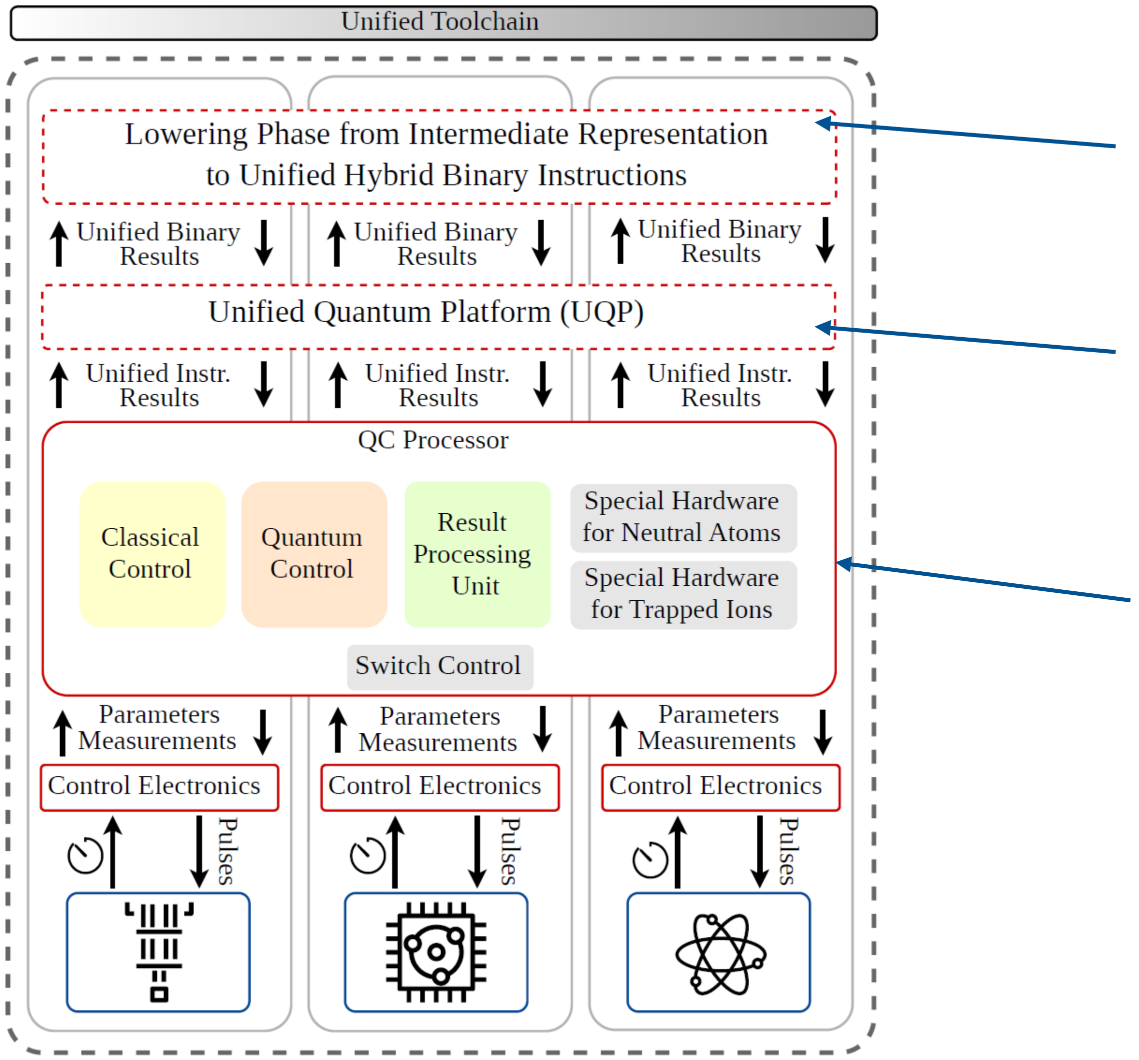
Targeting Layers


Software Stack
Quantum Algorithm
Quantum Programming Language
Quantum Compilation Toolchain
Executable Instructions
Quantum Control Processor
Quantum Instruction Set Architecture
Microarchitecture
Operations
Interface
Signal Generation and Readout Device
Pulses
Backend
Quantum Processing Unit (QPU)

Unified Quantum Platform – Overview



Unified Quantum Platform – Architecture



Runtime Environment

Instruction Set Architecture

Microarchitecture - FPGA

Unified Quantum Platform – Unified Runtime Environment

Map Quantum Intermediate Representation to Binary Instructions

- Takes care of scheduling instruction (“e.g., timing)
- Takes care of allocating memory (“e.g., registers)
- Generate the binary instructions
  - Map to hybrid (classic and quantum) instructions according to customized ISA

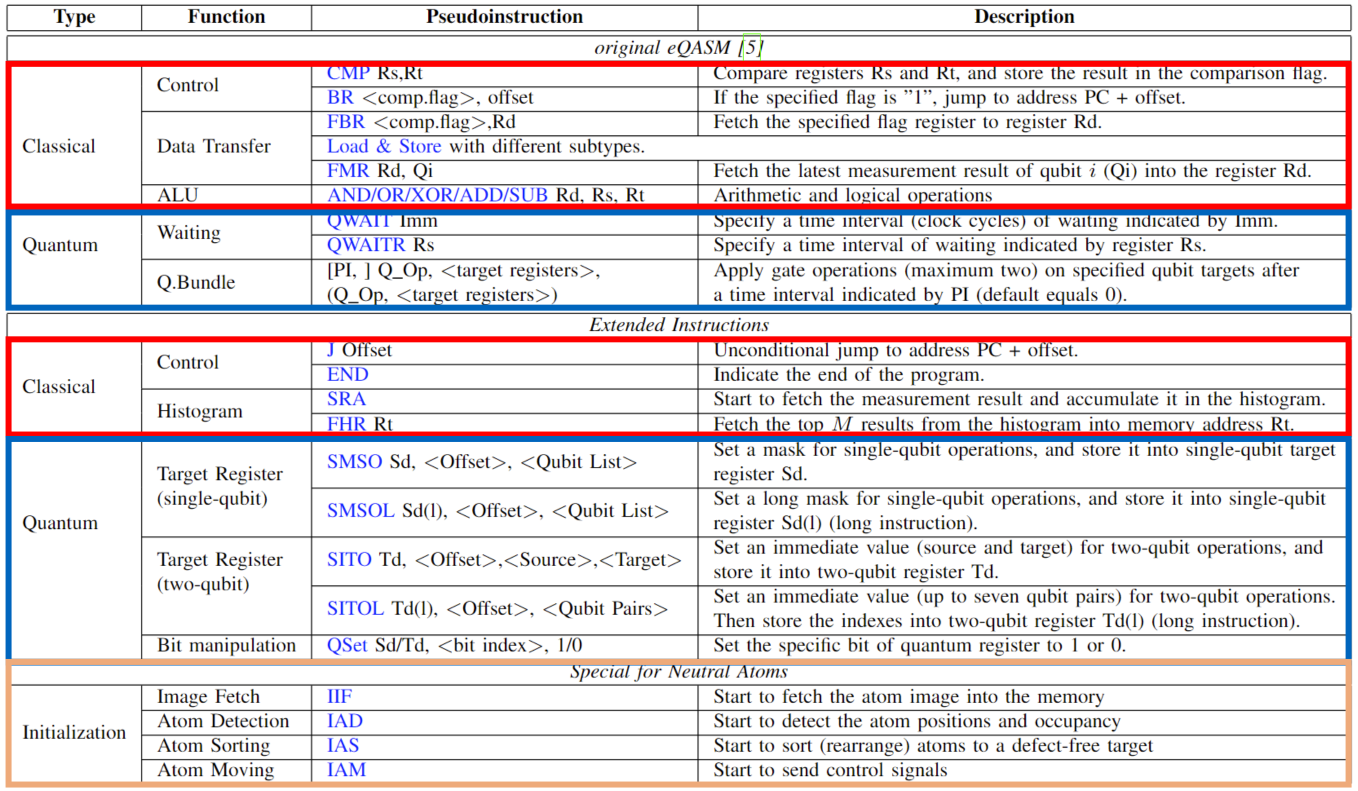
Work-in-progress

- Develop an internal representation that accommodates the hybrid instructions
- Investigate hybrid classical-quantum execution protocols/workflows.

Unified Quantum Platform – Unified Instruction Set Architecture

Current Features

- Hybrid Instruction Set
- Mixed length of instructions
  - Standard 32-bit instructions
  - Special Long instructions (128 bits)
- Fine-grained qubit control
- Mixed addressing mode: Immediate & Sliding Mask
- Support superconducting qubits & neutral atoms



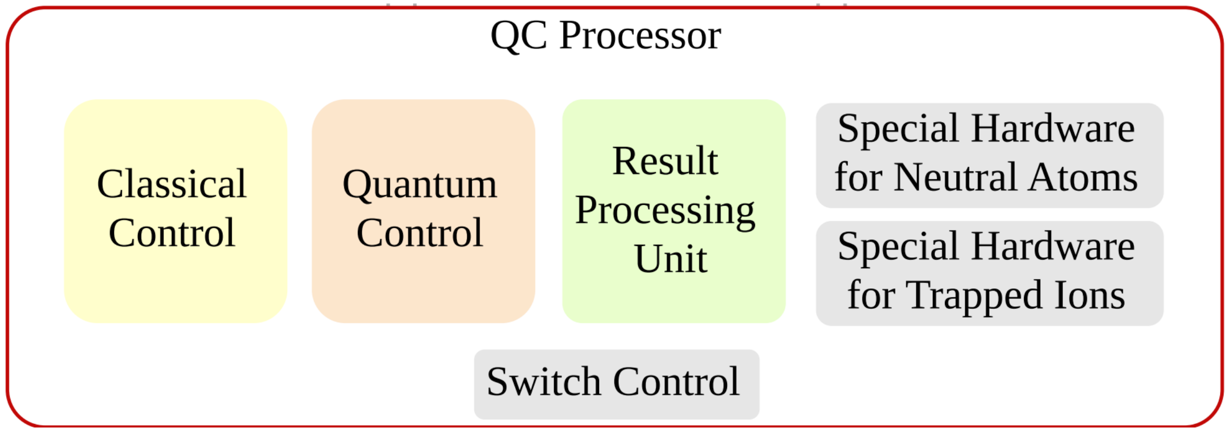
Type	Function	Pseudoinstruction	Description
original eQASM [5]			
Classical	Control	CMP Rs,Rt	Compare registers Rs and Rt, and store the result in the comparison flag.
		BR <comp.flag>, offset	If the specified flag is "1", jump to address PC + offset.
	Data Transfer	FBR <comp.flag>,Rd	Fetch the specified flag register to register Rd.
		Load & Store with different subtypes.	
		FMR Rd, Qi	Fetch the latest measurement result of qubit i (Qi) into the register Rd.
	ALU	AND/OR/XOR/ADD/SUB Rd, Rs, Rt	Arithmetic and logical operations
Quantum	Waiting	QWAIT Imm	Specify a time interval (clock cycles) of waiting indicated by Imm.
		QWAITR Rs	Specify a time interval of waiting indicated by register Rs.
	Q.Bundle	[PI,] Q_Op, <target registers>, (Q_Op, <target registers>)	Apply gate operations (maximum two) on specified qubit targets after a time interval indicated by PI (default equals 0).
Extended Instructions			
Classical	Control	J Offset	Unconditional jump to address PC + offset.
		END	Indicate the end of the program.
	Histogram	SRA	Start to fetch the measurement result and accumulate it in the histogram.
		FHR Rt	Fetch the top M results from the histogram into memory address Rt.
Quantum	Target Register (single-qubit)	SMSO Sd, <Offset>, <Qubit List>	Set a mask for single-qubit operations, and store it into single-qubit target register Sd.
		SMSOL Sd(l), <Offset>, <Qubit List>	Set a long mask for single-qubit operations, and store it into single-qubit register Sd(l) (long instruction).
	Target Register (two-qubit)	SITO Td, <Offset>,<Source>,<Target>	Set an immediate value (source and target) for two-qubit operations, and store it into two-qubit register Td.
		SITOL Td(l), <Offset>, <Qubit Pairs>	Set an immediate value (up to seven qubit pairs) for two-qubit operations. Then store the indexes into two-qubit register Td(l) (long instruction).
	Bit manipulation	QSet Sd/Td, <bit index>, 1/0	Set the specific bit of quantum register to 1 or 0.
Special for Neutral Atoms			
Initialization	Image Fetch	IIF	Start to fetch the atom image into the memory
	Atom Detection	IAD	Start to detect the atom positions and occupancy
	Atom Sorting	IAS	Start to sort (rearrange) atoms to a defect-free target
	Atom Moving	IAM	Start to send control signals

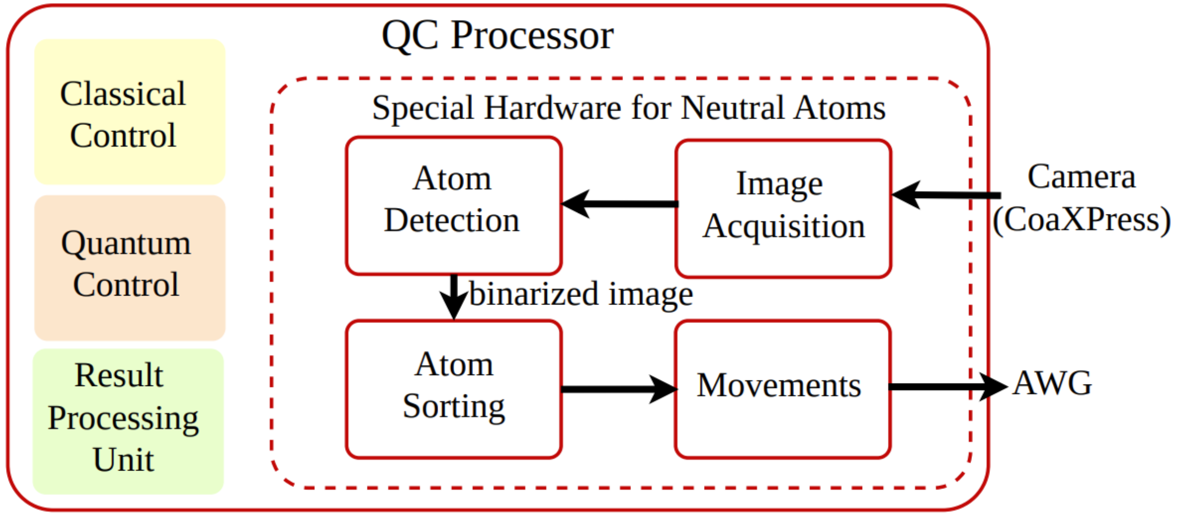
Unified Quantum Platform – Unified Quantum Control Processor

Features

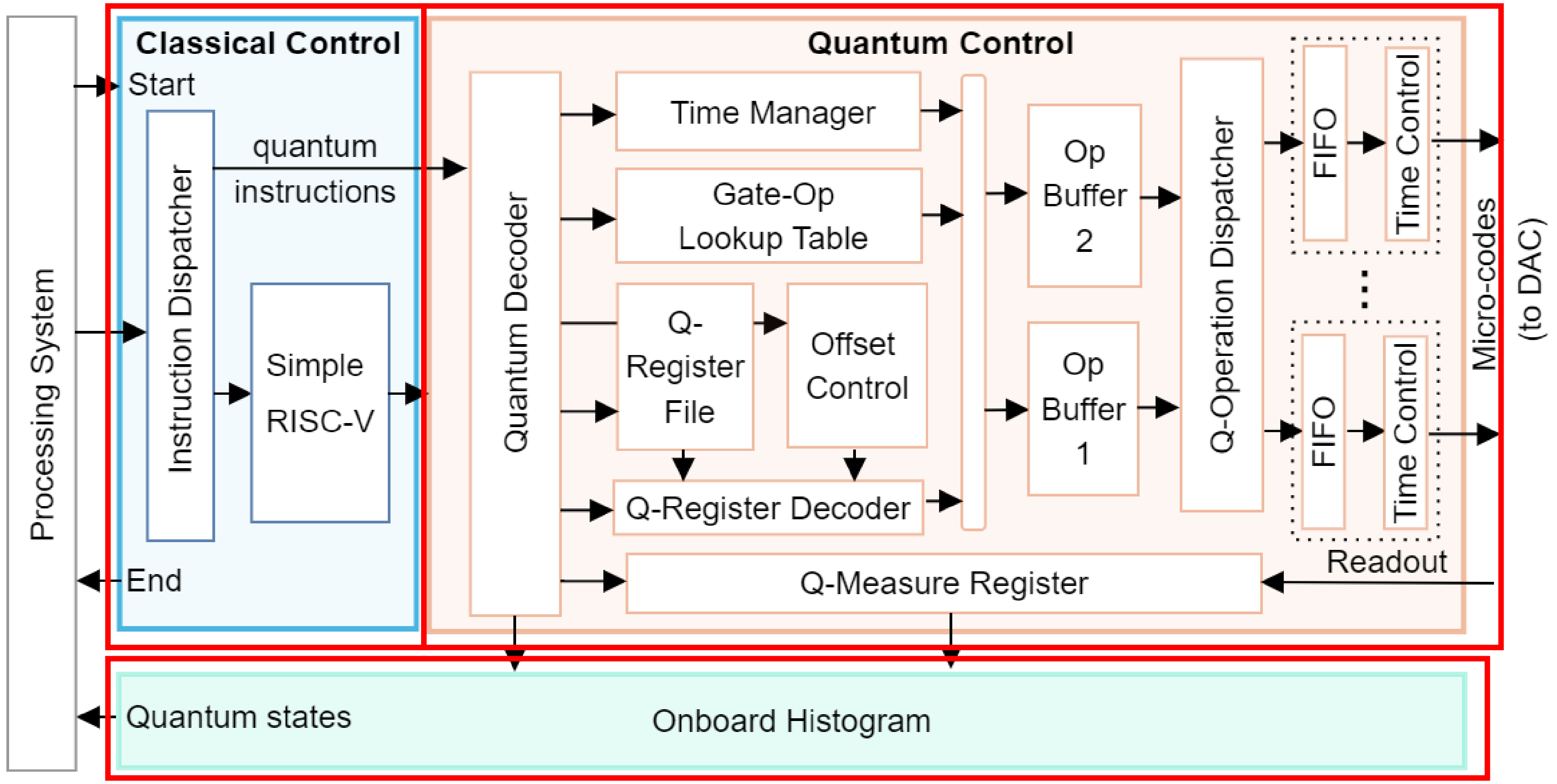
- Built on HiSEP-Q (A Highly Scalable and Efficient Quantum Control Processor for Superconducting Qubits)
  - QCP designed for superconducting qubits
- Technology-shared logics + Specialized acceleration block
- Switch control to change the modality

Example with Neutral-Atom Application





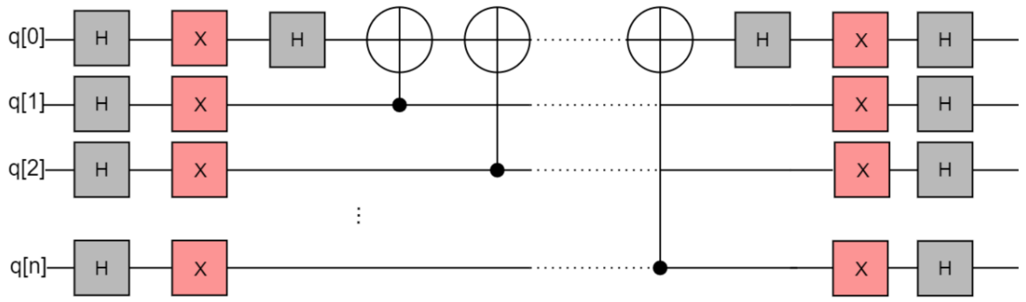
Microarchitecture – Processing Core

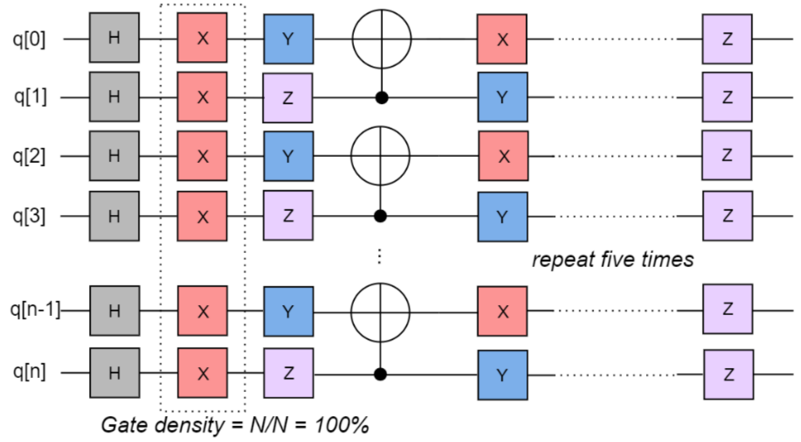


Evaluation - Experiment Setup

Dataset (Quantum Circuits)

- Munich Quantum Toolkit (MQT) Bench – Workflow experiments
- Real quantum circuit: Grover's operator (GO) & Synthetic circuits – QCP experiments

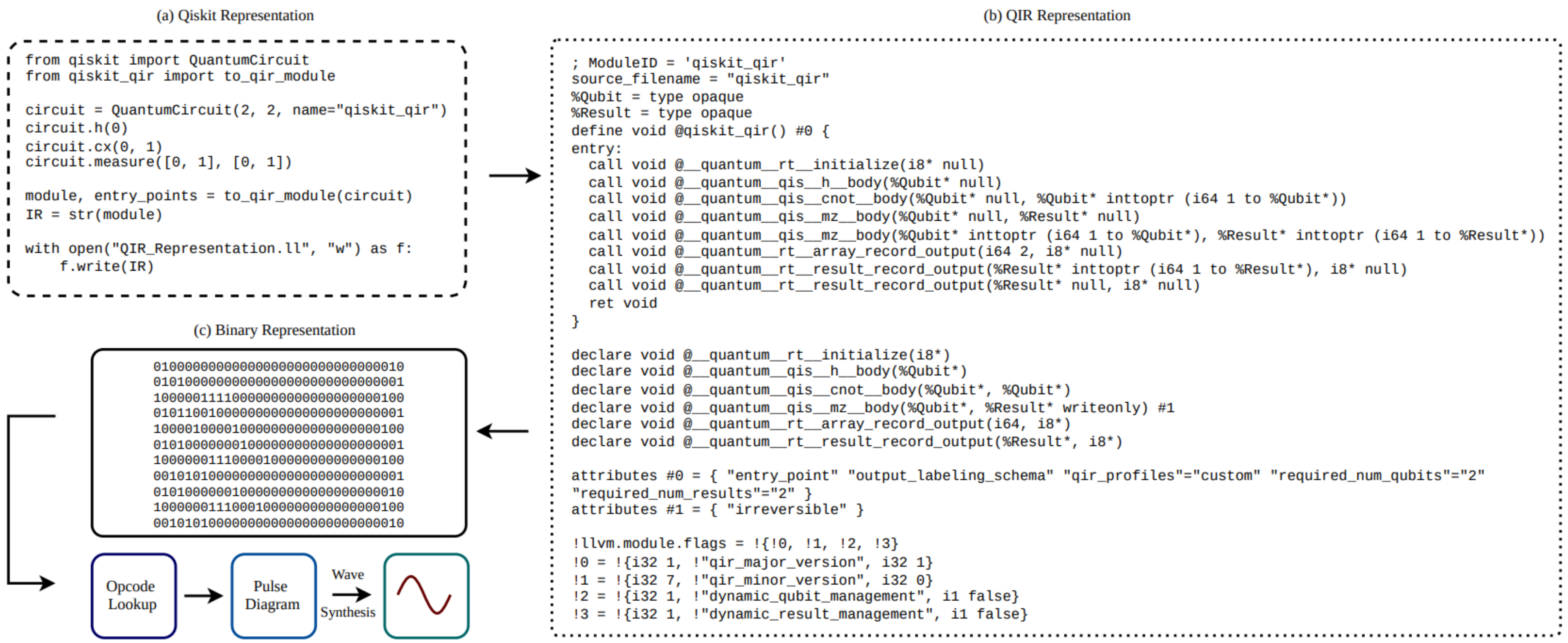




Hardware Setup

- Operating System: Linux
- CPU: 13th Gen Intel(R) Core(TM) i9- 13900HX
- FPGA : Xilinx ZCU216 & Pynq Z2

Evaluation - Workflow Verification I



(a) Qiskit Representation

```python
from qiskit import QuantumCircuit
from qiskit_qir import to_qir_module

circuit = QuantumCircuit(2, 2, name="qiskit_qir")
circuit.h(0)
circuit.cx(0, 1)
circuit.measure([0, 1], [0, 1])

module, entry_points = to_qir_module(circuit)
IR = str(module)

with open("QIR_Representation.ll", "w") as f:
    f.write(IR)
```

(b) QIR Representation

```
; ModuleID = 'qiskit_qir'
source_filename = "qiskit_qir"
%Qubit = type opaque
%Result = type opaque
define void @qiskit_qir() #0 {
entry:
  call void @__quantum__rt__initialize(i8* null)
  call void @__quantum__qis__h__body(%Qubit* null)
  call void @__quantum__qis__cnot__body(%Qubit* null, %Qubit* inttoptr (i64 1 to %Qubit*))
  call void @__quantum__qis__mz__body(%Qubit* null, %Result* null)
  call void @__quantum__qis__mz__body(%Qubit* inttoptr (i64 1 to %Qubit*), %Result* inttoptr (i64 1 to %Result*))
  call void @__quantum__rt__array_record_output(i64 2, i8* null)
  call void @__quantum__rt__result_record_output(%Result* inttoptr (i64 1 to %Result*), i8* null)
  call void @__quantum__rt__result_record_output(%Result* null, i8* null)
  ret void
}

declare void @__quantum__rt__initialize(i8*)
declare void @__quantum__qis__h__body(%Qubit*)
declare void @__quantum__qis__cnot__body(%Qubit*, %Qubit*)
declare void @__quantum__qis__mz__body(%Qubit*, %Result* writeonly) #1
declare void @__quantum__rt__array_record_output(i64, i8*)
declare void @__quantum__rt__result_record_output(%Result*, i8*)

attributes #0 = { "entry_point" "output_labeling_schema" "qir_profiles"="custom" "required_num_qubits"="2"
"required_num_results"="2" }
attributes #1 = { "irreversible" }

!llvm.module.flags = !{!0, !1, !2, !3}
!0 = !{i32 1, !"qir_major_version", i32 1}
!1 = !{i32 7, !"qir_minor_version", i32 0}
!2 = !{i32 1, !"dynamic_qubit_management", i1 false}
!3 = !{i32 1, !"dynamic_result_management", i1 false}
```

(c) Binary Representation

```
01000000000000000000000000000010
01010000000000000000000000000001
10000011110000000000000000000100
01011001000000000000000000000001
10000100001000000000000000000100
01010000000100000000000000000001
10000001110000100000000000000100
00101010000000000000000000000001
01010000001000000000000000000010
10000001110001000000000000000100
00101010000000000000000000000010
```

Evaluation - Workflow Verification II

Mapping Workflow

a) Quantum circuits representing bell state in qiskit

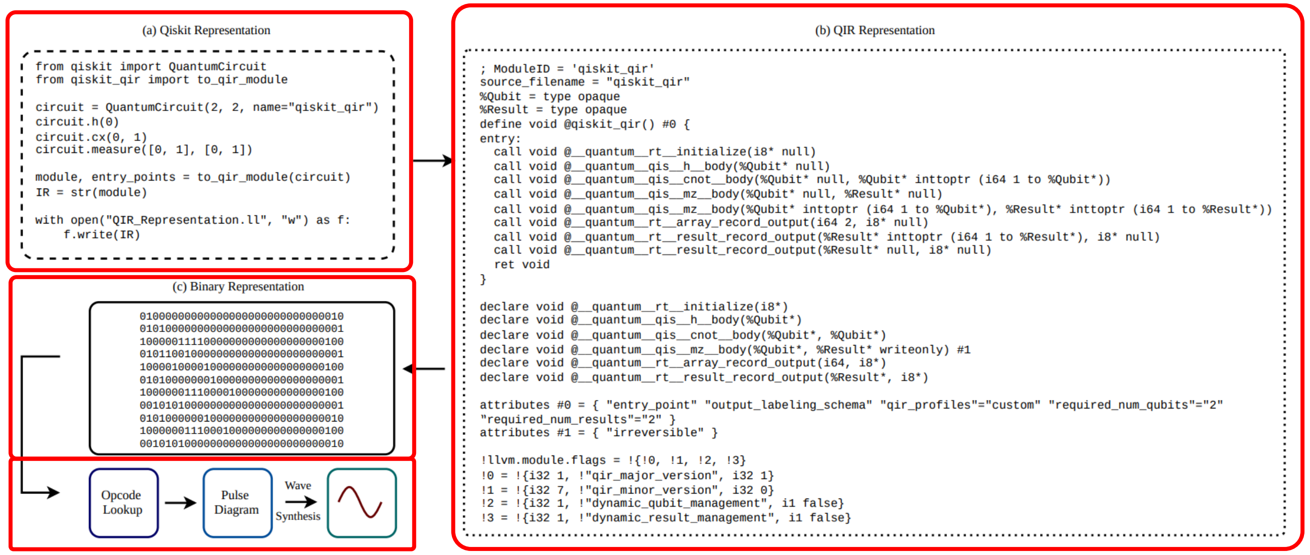
b) Corresponding QIR implementation

- Each instruction is represented by an external function call to the backend runtime library

c) Binary instructions

d) Waveform generation

- By the control logics on FPGA


(a) Qiskit Representation

```
from qiskit import QuantumCircuit
from qiskit_qir import to_qir_module

circuit = QuantumCircuit(2, 2, name="qiskit_qir")
circuit.h(0)
circuit.cx(0, 1)
circuit.measure([0, 1], [0, 1])

module, entry_points = to_qir_module(circuit)
IR = str(module)

with open("QIR_Representation.ll", "w") as f:
    f.write(IR)
```

(b) QIR Representation

```
; ModuleID = 'qiskit_qir'
source_filename = "qiskit_qir"
%Qubit = type opaque
%Result = type opaque
define void @qiskit_qir() #0 {
entry:
  call void @__quantum__rt__initialize(i8* null)
  call void @__quantum__qis__h__body(%Qubit* null)
  call void @__quantum__qis__cnot__body(%Qubit* null, %Qubit* inttoptr (i64 1 to %Qubit*))
  call void @__quantum__qis__mz__body(%Qubit* null, %Result* null)
  call void @__quantum__qis__mz__body(%Qubit* inttoptr (i64 1 to %Qubit*), %Result* inttoptr (i64 1 to %Result*))
  call void @__quantum__rt__array_record_output(i64 2, i8* null)
  call void @__quantum__rt__result_record_output(%Result* inttoptr (i64 1 to %Result*), i8* null)
  call void @__quantum__rt__result_record_output(%Result* null, i8* null)
  ret void
}

declare void @__quantum__rt__initialize(i8*)
declare void @__quantum__qis__h__body(%Qubit*)
declare void @__quantum__qis__cnot__body(%Qubit*, %Qubit*)
declare void @__quantum__qis__mz__body(%Qubit*, %Result* writeonly) #1
declare void @__quantum__rt__array_record_output(i64, i8*)
declare void @__quantum__rt__result_record_output(%Result*, i8*)

attributes #0 = { "entry_point" "output_labeling_schema" "qir_profiles"="custom" "required_num_qubits"="2"
"required_num_results"="2" }
attributes #1 = { "irreversible" }

!llvm.module.flags = !{!0, !1, !2, !3}
!0 = !{i32 1, !"qir_major_version", i32 1}
!1 = !{i32 7, !"qir_minor_version", i32 0}
!2 = !{i32 1, !"dynamic_qubit_management", i1 false}
!3 = !{i32 1, !"dynamic_result_management", i1 false}
```

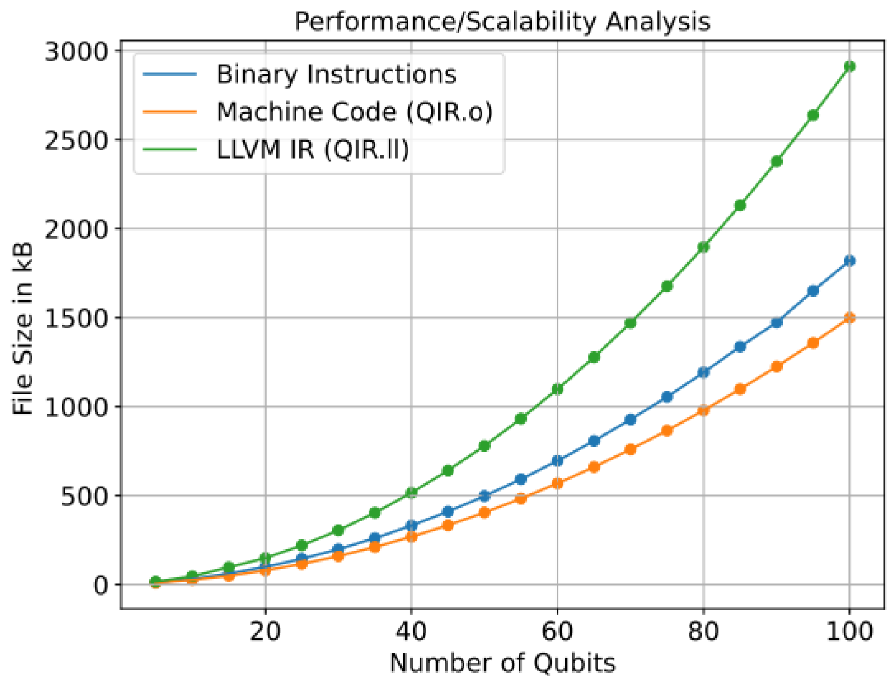
(c) Binary Representation

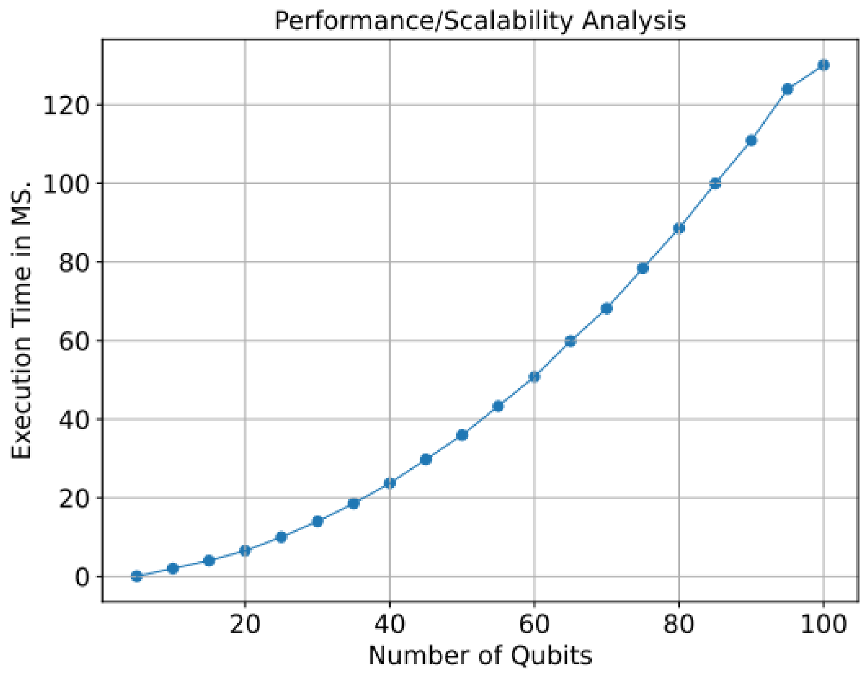
```
01000000000000000000000000000010
01010000000000000000000000000001
10000011110000000000000000000100
01011001000000000000000000000001
10000100001000000000000000000100
01010000000100000000000000000001
10000001110000100000000000000100
00101010000000000000000000000001
01010000001000000000000000000010
10000001110001000000000000000100
00101010000000000000000000000010
```

Qiskit Representation	Binary Instruction Representation
N/A	01000000000000000000000000000010 - Execution environment initialization
circuit.h(0)	01010000000000000000000000000001 - Memory instruction 10000011110000000000000000000100 - Hadamard operation
circuit.cx(0, 1)	01011001000000000000000000000001 - Memory instruction 10000100001000000000000000000100 - CNOT operation
circuit.measure([0, 1], [0, 1])	01010000000100000000000000000001 - Memory instruction 10000001110000100000000000000100 - First qubit measurement operation 00101010000000000000000000000001 - Fetch last measurement 01010000001000000000000000000010 - Memory instruction 10000001110001000000000000000100 - Second qubit measurement operation 00101010000000000000000000000010 - Fetch last measurement

Evaluation - Memory and Time Performance

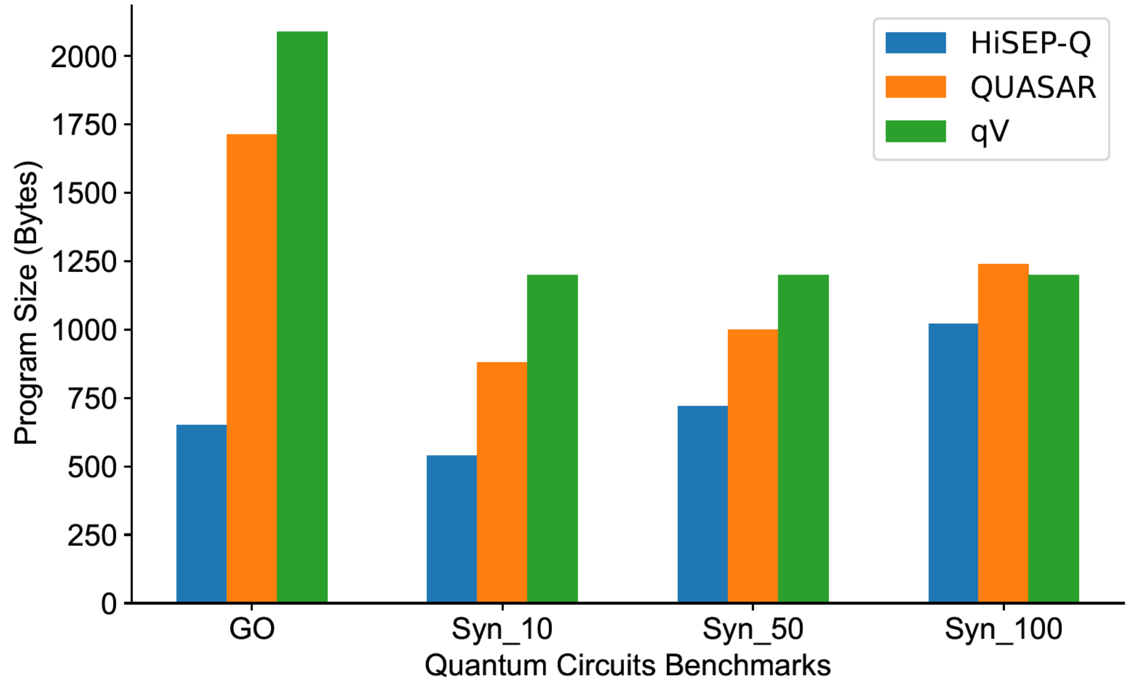
- Super-linearly scales with the number of qubits (size of quantum codes)
- Software infrastructure can expand to accommodate this growth **without facing exponential increases in resource**
- Guarantee for the usability in **larger and more complex** quantum algorithms





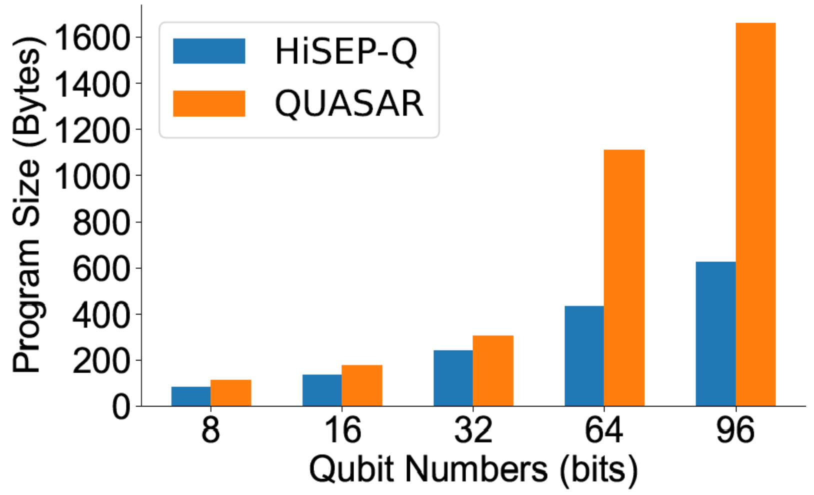
Evaluation – QISA Efficiency

- Comparing with state-of-the-art works [2] using 100 qubits (exclude eQASM, as 100 qubits are not supported).
- Four datasets: GO and synthetic quantum circuits with varied gate density.
  - Gate density : (the degree of the available gates implemented in the circuits at the same time)
- 62% improvement in real quantum circuit
- Average 28% improvement in synthetic circuits

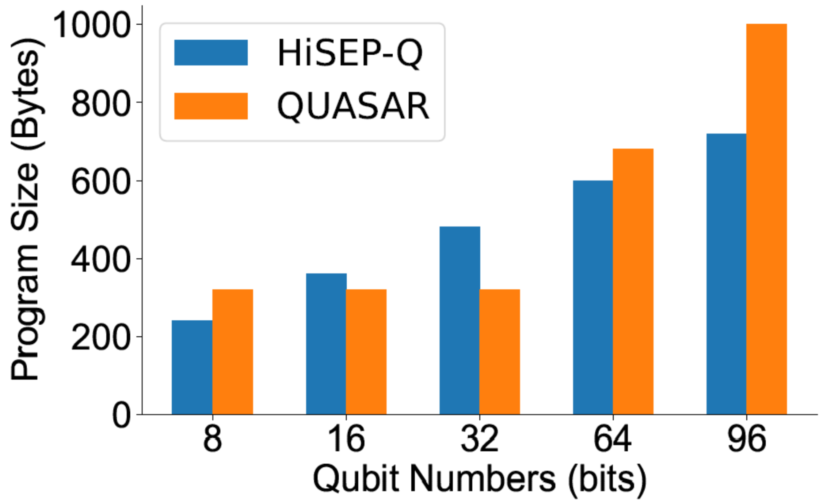


Evaluation – QISA Scalability

- Scalability performance
- Two datasets: GO and synthetic quantum circuits with 50% gate density.
- HiSEP-Q: Logarithmic increase
  - Trend of increase is significantly lower than QUASAR



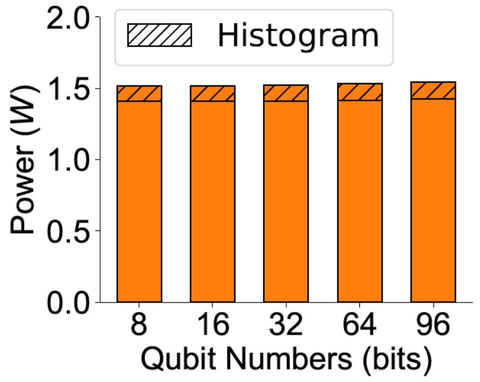
(a) GO



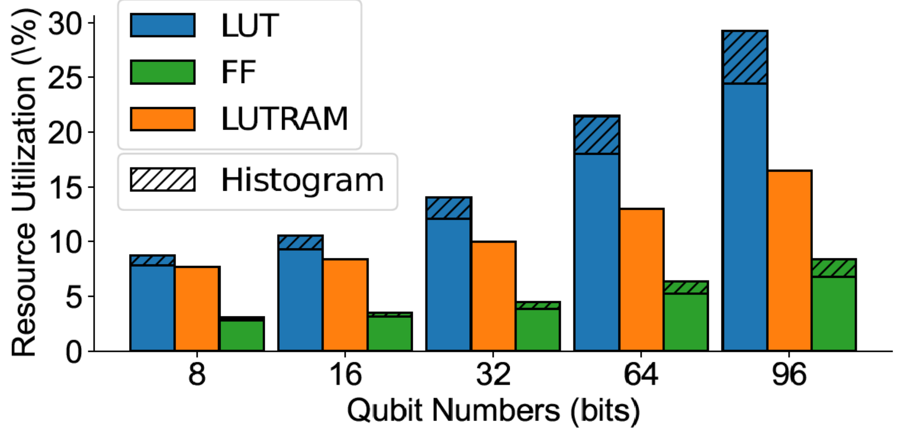
(b) Syn_50

Evaluation - Microarchitecture

- Zynq SoC implementation
- Constant power consumption
- Negligible overhead of onboard histogram
- Logarithmically increased resource utilization with number of qubits
  - Only 30% of LUT and 15% of LUTRAM, when testing with 96 qubits



(a) Power



(b) Resource Utilization

Conclusion & Outlook

Conclusion

- We implemented an abstraction layer needed to realize a unified quantum platform
  - A novel unified runtime library
  - A unified hybrid ISA and QCP
- Comprehensive workflow verification
- The first idea of a **unified and open quantum platform** and to implement it **within a tight HPCQC integration** setup

Future work:

- Extension & optimization of the ISA and corresponding QCP
- Extension of the execution/runtime environment.

Contact:

Xiaorang Guo

xiaorang.guo@tum.de

https://www.ce.cit.tum.de/caps/startseite/