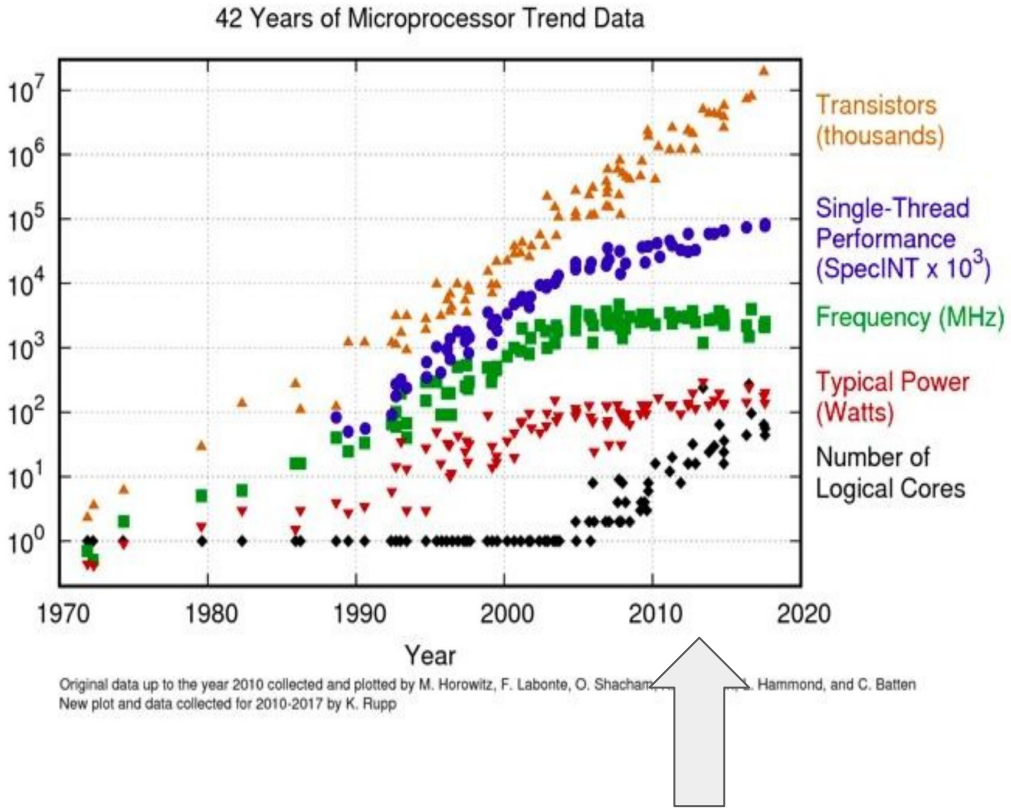
A Formal Verification Model and Language for the Correctness of Programs on NA-QPUs

Daniel Vonk

Post-CMOS Computing and the Rise of Quantum Computing

- Advancements in processing speed have traditionally been underpinned by the scalability of microprocessors–specifically transistor shrinking, i.e. *Moore's Law*.
- Since ~2015, however, the era of *diminishing returns* has reigned.
    - Frequency plateau
    - $/transistor plateau
    - Quantum leakage/tunneling engineering challenges



Era of "diminishing returns" begins

Post-CMOS Computing and the Rise of Quantum Computing

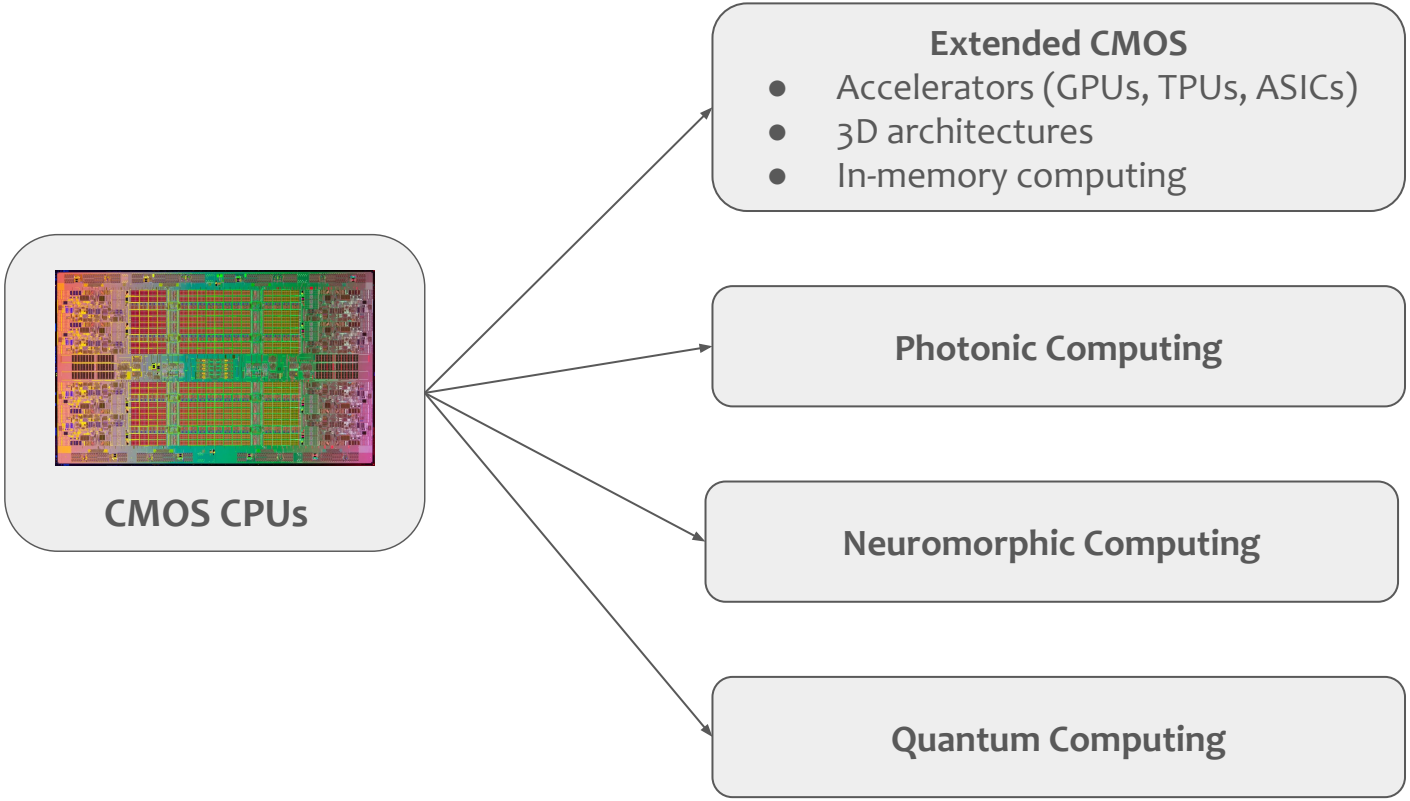


Image Credit: [1]

Post-CMOS Computing and the Rise of Quantum Computing

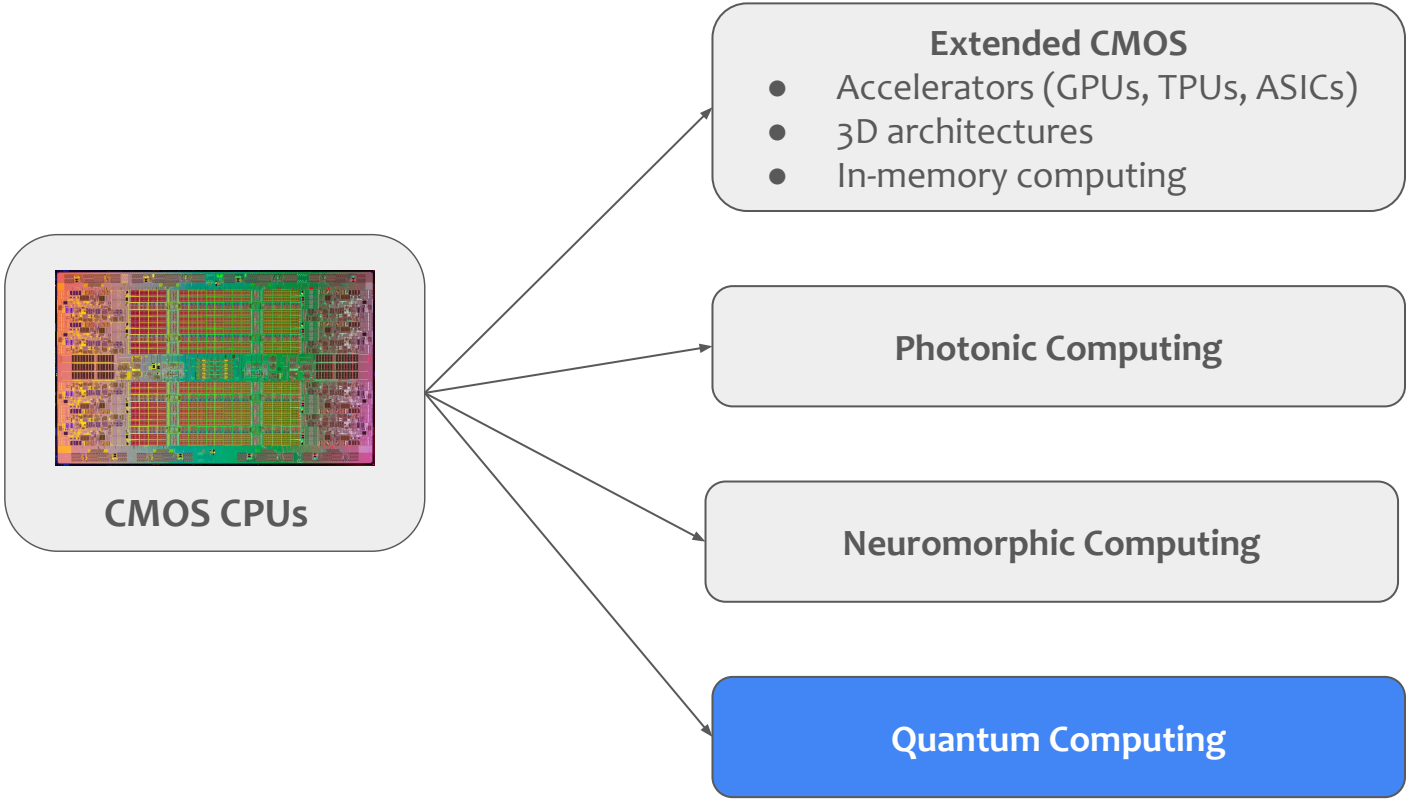


Image Credit: [1]

The Rise of Quantum Computing

Large Industry Investment

IBM Google Microsoft aws QUANTINUUM

Increasing Hardware Development

Governmental Funding Programmes

QUANTUM FLAGSHIP

Munich Quantum Valley

⟨quantum|gov⟩

The Rise of Quantum Computing

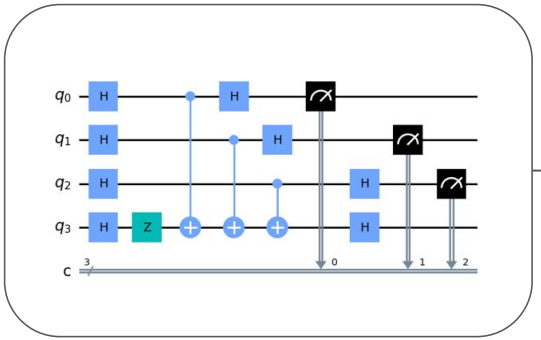
- Quantum computers aren't just "better CPUs" – they utilize *superposition*, *entanglement* and *interference* to solve some problems **exponentially** faster:
  - Shor's algorithm: exponentially faster integer factorization
  - HHL algorithm: exponentially faster linear solve
  - Grover's algorithm: quadratically faster search
  - Quantum Amplitude Estimation: quadratically faster expected value estimation

The Rise of Quantum Computing

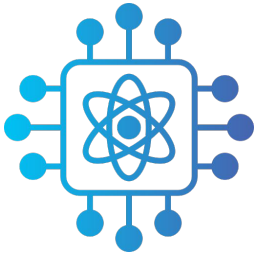
- Quantum computers aren't just "better CPUs" – they utilize *superposition*, *entanglement* and *interference* to solve some problems **exponentially** faster:
  - Shor's algorithm: exponentially faster integer factorization
  - HHL algorithm: exponentially faster linear solve
  - Grover's algorithm: quadratically faster search
  - Quantum Amplitude Estimation: quadratically faster expected value estimation

Quantum Computing offers great potential for many domains e.g. Machine Learning, Financial Modelling and Chemistry.

Quantum Compilation (briefly)



A quantum program can be written as a circuit (digital sequence of gates) and represents a **unitary transformation** on an initial state.



The quantum computer implements the transformation using its native hardware.

Measurement collapses the state into a **classical bitstring** (e.g. `1010`). The collapse is probabilistic so the process must be repeated many times (*shots*) to gain an accurate distribution.

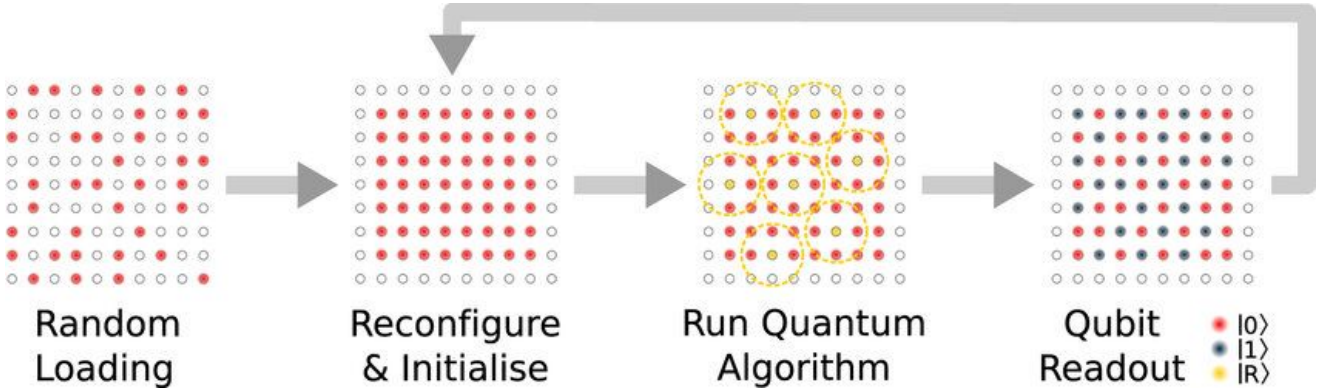
Types of Quantum Computer

- Trapped ion qubits
  - Qubits encoded in state of charged atoms and trapped electromagnetically.
  - Long coherence times (<1min) and high-fidelity gates.
  - Challenges: slow gate times, hard to scale number of qubits.
- Neutral-Atom qubits
  - Individual neutral atoms trapped optically. Gates applied through Rydberg interactions.
  - Highly scalable (>1000s qubits), long coherence times and reconfigurable layouts.
  - Challenges: complex laser control, precise optical systems required.
- Superconducting qubits
  - Qubits built from junction circuits cooled to millikelvin temperatures.
  - Fast gate times (~10-100ns)
  - Most mature platform. Used by IBM and Google.
  - Challenges: coherence times, fixed connectivity.

Types of Quantum Computer

- Trapped ion qubits
  - Qubits encoded in state of charged atoms and trapped electromagnetically.
  - Long coherence times (<1min) and high-fidelity gates.
  - Challenges: slow gate times, hard to scale number of qubits.
- Neutral-Atom qubits
  - Individual neutral atoms trapped optically. Gates applied through Rydberg interactions.
  - Highly scalable (>1000s qubits), long coherence times and reconfigurable layouts.
  - Challenges: complex laser control, precise optical systems required.
- Superconducting qubits
  - Qubits built from junction circuits cooled to millikelvin temperatures.
  - Fast gate times (~10-100ns)
  - Most mature platform. Used by IBM and Google.
  - Challenges: coherence times, fixed connectivity.

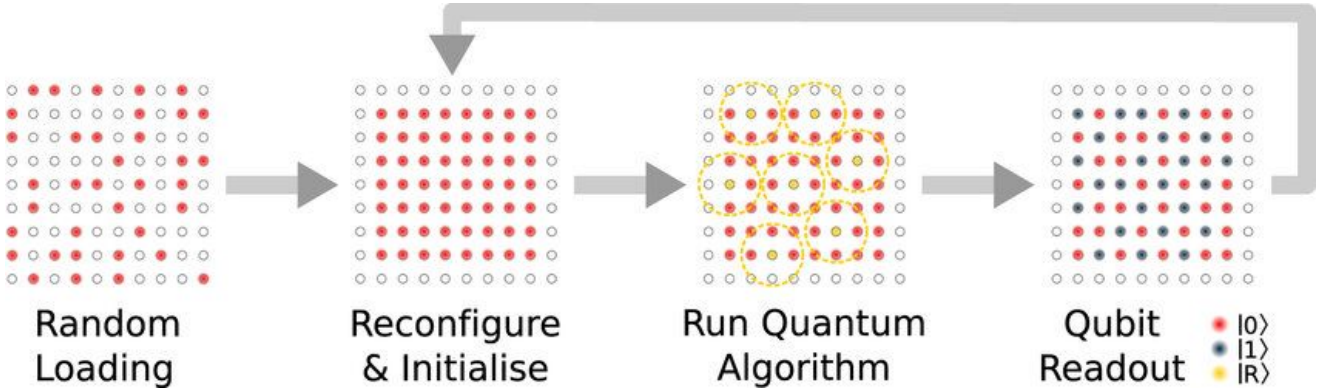
Neutral-Atom Quantum Computing



- Neutral atoms (Rb,Cs,Yb) are supercooled and loaded into a vacuum chamber.
- **Rabi** laser pulses and **Rydberg** laser pulses perform the 1Q/2Q gates respectively.
- **AOD/SLM** lasers used for reconfigurable atom movement.
- The neutral atoms remain in a coherent state for ~minutes.
- An (optical) camera is used to measure the results of the algorithm run by the QPU.

Image Credit: "Rydberg atom quantum technologies", CS Adams et al., Physics B

Neutral-Atom Quantum Computing



Problem #1: Operating the QPU correctly requires implementing precise control signals for several systems (lasers, traps, camera etc.) and subtle bugs in a compiled program can cause the entire run to become decoherent.

Image Credit: "Rydberg atom quantum technologies", CS Adams et al., Physics B

Neutral-Atom Quantum Computing

- Neutral-atom QPUs are not standardized: different vendors have different architectures.
- NA QPUs are likewise under rapid development and the architectures subject to change.

Image Credit: [3],[4],[5]

Neutral-Atom Quantum Computing

Problem #2: There is no uniform representation for compiled quantum programs across different vendors which allows for iterative architectural improvements (i.e. modularity).

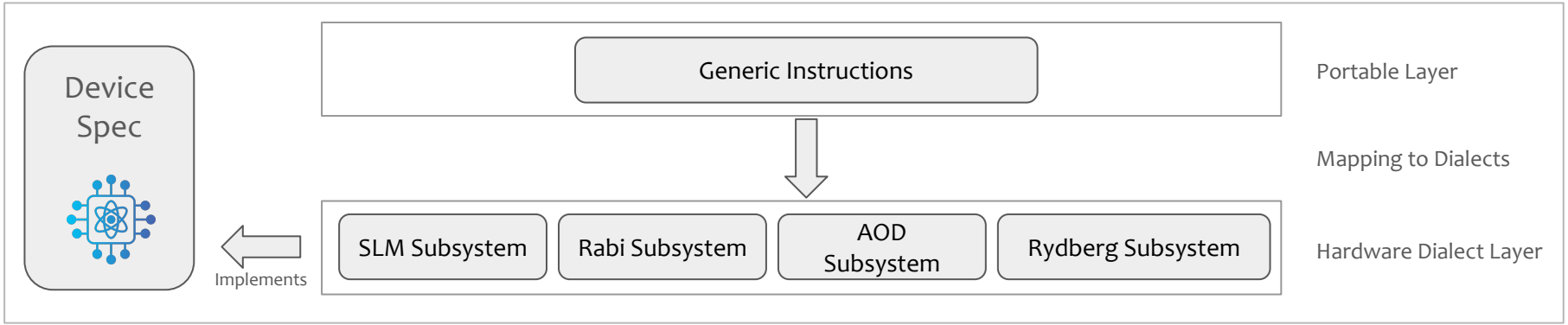
Image Credit: [3],[4],[5]

Problem Statement

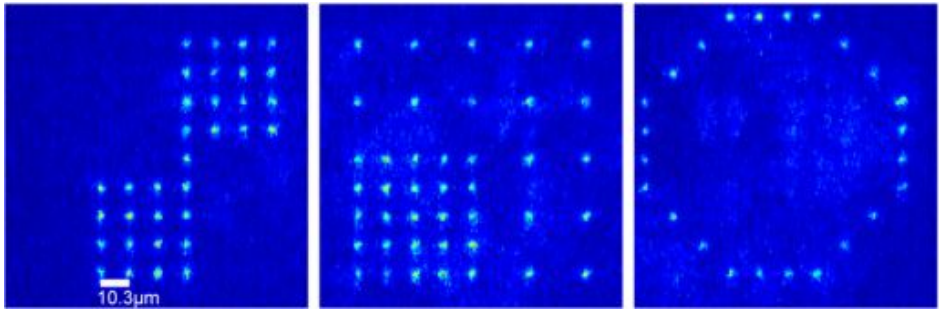
"How can an instruction set be built for neutral-atom QPUs such that it supports **multiple architectures** and allows for future **extensibility**? How can programs in this language be **verified for correctness** in a principled way?"

The AtomGuard ISA

Contribution: **AtomGuard ISA**
A unified framework for NA QPUs to support **modularity, reconfiguration** and **formal verification.**



The AtomGuard ISA: Portable Layer



- Key Idea: **Generic** set of operations that all hardware must support
  - Atom movement, 1Q/2Q gate application, qubit allocation/release etc.
- No reference to physical device operations
  - E.g. Rydberg radius, movement constraints
- Correctness of high-level algorithmic properties can already be verified at this layer

Image Credit: [6]

The AtomGuard ISA: Portable Layer

- Instructions are written in JSON format and reference the device specified in the DeviceSpec.
- No explicit timing specifications – (partial) ordering by IDs
- Other instructions: `init`, `rydberg`, `1q`, `barrier`. See thesis for details.

```
{
    "id": 2, "type": "rearrange", "depends": [1],
      "begin_locs": [{ "qubit":"q0",
"zone":"store0", "array":0, "row":12,
"col":3 }, ... ],
"end_locs":
      [{ "qubit":"q0", "zone":"ent0",
"array":1, "row":0, "col":0 }, ... ],
"transport": "aod0"
}
```

Figure: Example movement instruction

Image Credit: [6]

The AtomGuard ISA: Dialects

- Key Idea: Each dialect describes the semantics of a particular **hardware target**
- We include 4 dialects as a starting point
  - AOD
  - SLM
  - Rydberg
  - Rabi
- Each instruction in a dialect represents a low-level control signal for the hardware component
- Dialects are parameterized by the machine specification, which provides
  - Grid geometry
  - AOD speed constraints
  - Gate durations
  - …

Image Credit: [6]

The AtomGuard ISA: Dialects

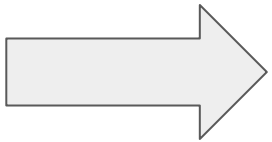
Constructor	Field	Type	Purpose
`slm/activate`			Activate the AOD laser at a specific intersection point
	`row`		The row of the location to activate
	`col`		The column of the location to activate
	`duration`	$\mathbb{R}_{\geq 0}$	The duration of ramp-up time
`slm/deactivate`			Deactivate the AOD laser at a specific intersection point
	`row`		The row of the location to deactivate
	`col`		The column of the location to deactivate

The AtomGuard ISA: Dialects

- Each portable instruction must now be replaced with one or more dialect operations which reflect
    - QPU's available primitives
    - Topology
    - Timing model
    - Physical constraints
- We call this "lowering"

The AtomGuard ISA: Dialects

```
{ "id": 3, "type":
"rearrange",
"begin_locs": [...],
"end_locs": [...],
"transport": "aod0" }
```



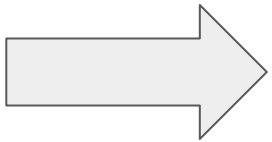
Each portable instruction is lowered to the dialects by the compiler

```
{
  "lowerings": [{
    "id": 30,
    "dialect": [“aod”, “slm”],
    "ops": [
      { "op": "aod/activate",    "tone_id": 0, "fx": 98.0,   "fy": 12.5, "power": …},
      { "op": "aod/activate",    "tone_id": 1, "fx": 101.0, "fy": 12.5, "power": …},
      { "op": "aod/chirp",       "tone_id": 0,
        "from": [98.0, 12.5], "to": [108.0, 14.0], "duration": 600
      },
      { "op": "aod/chirp",       "tone_id": 1,
        "from": [101.0, 12.5], "to": [111.0, 14.0], "duration": 60000
      },
      { "op": "aod/ampl_ramp",   "tone_id": 0, "p0": 0.20, "p1": 0.00, "duration": 100 },
      { "op": "aod/ampl_ramp",   "tone_id": 1, "p0": 0.20, "p1": 0.00, "duration": 100 },
      { "op": "aod/deactivate", "tone_id": 0 },
      { "op": "aod/deactivate", "tone_id": 1 },

      { "op": "slm/activate",    "row": 0, "col": 0, "duration": 500 },
      { "op": "slm/activate",    "row": 0, "col": 1, "duration": 500 },
      { "op": "slm/deactivate", "row": 12, "col": 3 },
      { "op": "slm/deactivate", "row": 12, "col": 4 }
    ],

    "contracts": [
      "compatible2D","vmax","aodSlmHandoff","noPulseDuringMove"
    ]
  }]
}
```

Image Credit: [6]

The AtomGuard ISA: Dialects

```
{ "id": 3, "type":
"rearrange",
"begin_locs": [...],
"end_locs": [...],
"transport": "aod0" }
```



Each portable instruction is lowered to the dialects by the compiler

```
{
  "lowerings": [{
    "id": 30,
    "dialect": [“aod”, “slm”],
    "ops": [
      { "op": "aod/activate",    "tone_id": 0, "fx": 98.0,   "fy": 12.5, "power": …},
      { "op": "aod/activate",    "tone_id": 1, "fx": 101.0, "fy": 12.5, "power": …},
      { "op": "aod/chirp",       "tone_id": 0,
        "from": [98.0, 12.5], "to": [108.0, 14.0], "duration": 600
      },
      { "op": "aod/chirp",       "tone_id": 1,
        "from": [101.0, 12.5], "to": [111.0, 14.0], "duration": 60000
      },
      { "op": "aod/ampl_ramp",   "tone_id": 0, "p0": 0.20, "p1": 0.00, "duration": 100 },
      { "op": "aod/ampl_ramp",   "tone_id": 1, "p0": 0.20, "p1": 0.00, "duration": 100 },
      { "op": "aod/deactivate", "tone_id": 0 },
      { "op": "aod/deactivate", "tone_id": 1 },

      { "op": "slm/activate",    "row": 0, "col": 0, "duration": 500 },
      { "op": "slm/activate",    "row": 0, "col": 1, "duration": 500 },
      { "op": "slm/deactivate", "row": 12, "col": 3 },
      { "op": "slm/deactivate", "row": 12, "col": 4 }
    ],

    "contracts": [
      "compatible2D","vmax","aodSlmHandoff","noPulseDuringMove"
    ]
  }]
}
```

Image Credit: [6]

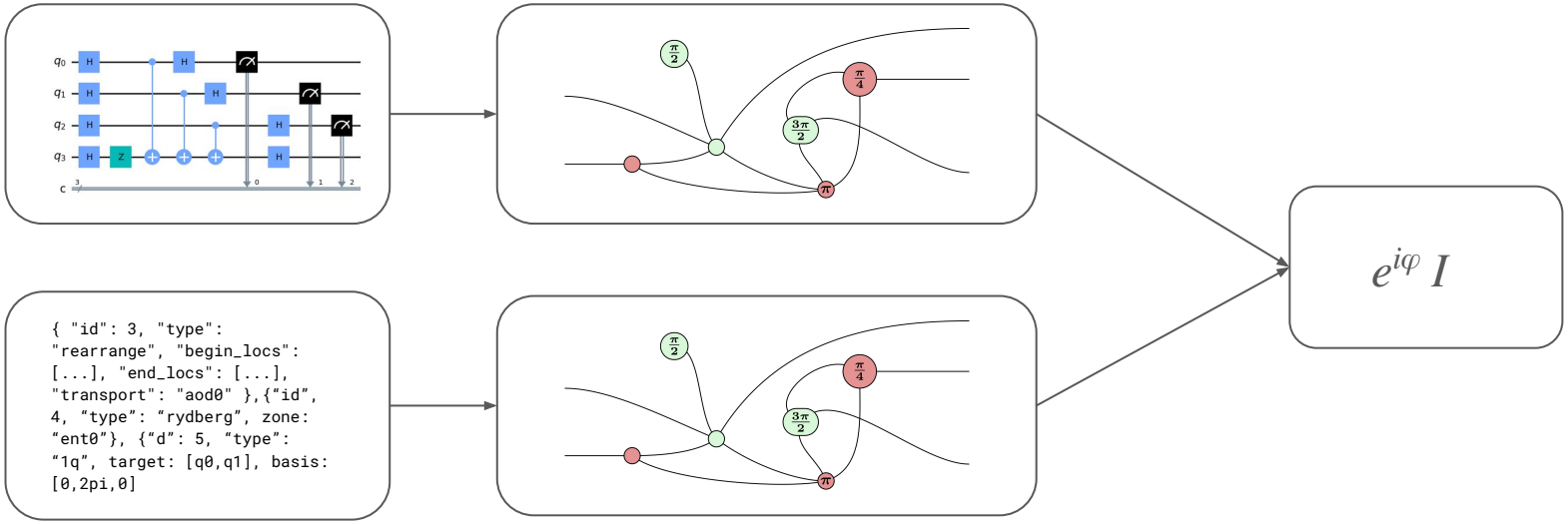
The AtomGuard ISA: Verification

- We would also like to reason about the correctness of these compiled instructions
- In AtomGuard, we do this on two levels
  - Verify the portable instructions are **semantically equivalent** to the input circuit
  - Verify the dialects satisfy the **hardware correctness** contracts

The AtomGuard ISA: Verification

- Semantic equivalence to the input circuit can be done simply thanks to the portable layer abstraction
  - There is a direct correspondence between portable instructions and circuit operations
- Equivalence of circuit diagrams can be efficiently checked through the ZX-calculus[9]
  - Goal: show $U_{comp} = e^{i\varphi} U_{in}$ for $\varphi \in \mathbb{C}$, i.e. functional equivalence

The AtomGuard ISA: Verification



Both source and target are converted into ZX-diagrams.

Target ZX diagram is reversed and the two are then composed into one diagram.

ZX operations are performed until identity is reached. This gives the decision procedure for equivalence.

The AtomGuard ISA: Verification

- We also want to verify hardware correctness checks or **contracts** e.g.
  - No laser pulses on the zone while a movement is active
  - Movements follow a physically plausible path and at correct velocity
  - A Rydberg pulse always follows a move into the entanglement zone
  - SLM activated directly after an AOD move into the trap
  - Rydberg and Rabi pulses never occur at the same time

```
"contracts": [
  "compatible2D","vmax","aodSlmHandoff","noPulseDuringMove"
]
```

Figure: Contracts specified for a lowering.

The AtomGuard ISA: Verification Technique

- Naive approach: build an operational interpreter[8] in Lean
- For each dialect and for each instruction, construct precise semantics:

$$\frac{t_{\text{start}} < t_{\text{end}} \qquad \text{pos}(q_0, t_{\text{start}}) = [x, y] \qquad \frac{\|[z,w]-[x,y]\|}{t_{\text{start}}-t_{\text{end}}} \leq v_{\max}}{(\text{move}(q_0, [x, y], [z, w], t_{\text{start}}, t_{\text{end}}), s) \Rightarrow \text{add\_path}(s, q_0, [x, y], [z, w], t_{\text{start}}, t_{\text{end}})}\ \text{MoveOK}$$

$$\frac{t_{\text{start}} < t_{\text{end}} \quad \text{pos}(q_0, t_{\text{start}}) = [x, y] \quad \frac{\|[z,w]-[x,y]\|}{t_{\text{start}}-t_{\text{end}}} > v_{\max}}{(\text{move}(q_0, [x, y], [z, w], t_{\text{start}}, t_{\text{end}}), s) \Rightarrow \text{Fault\_TooFast}}\ \text{Move2Fast}$$

$$\frac{\begin{array}{c} t_{\text{start}} < t_{\text{end}} \quad \text{pos}(q_0, t_{\text{start}}) = [x, y] \quad \frac{\|[z,w]-[x,y]\|}{t_{\text{start}}-t_{\text{end}}} \leq v_{\max} \\ q_0 \text{ scheduled already in } s \text{ at } [t_{\text{start}}, t_{\text{end}}) \end{array}}{(\text{move}(q_0, [x, y], [z, w], t_{\text{start}}, t_{\text{end}}), s) \Rightarrow \text{Fault\_MoveConflict}}\ \text{MoveConflict}$$

The AtomGuard ISA: Verification Technique

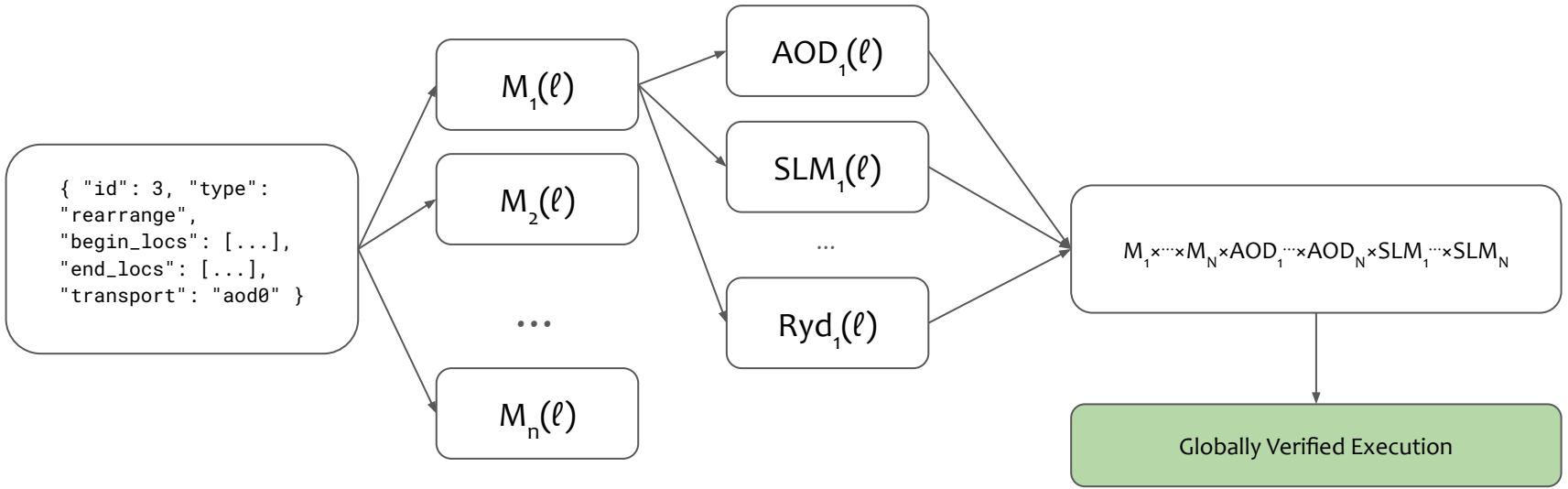
- A state is maintained by the verifier and **mutated** with each instruction
  - $\langle S_0 \rangle \rightarrow \langle S_1 \rangle \rightarrow \ldots \rightarrow \langle S_T \rangle$
  - The effect of each instruction is described precisely by the operational semantics
- But this has significant disadvantages!
  - Requires a single **global** state, which is **not modular**
  - Correctness checks must be done **sequentially**, not **parallelizable**
  - Writing this in Lean involves creating a single, large inductive type for the machine state with a mix of discrete and continuous properties. Invariants have to be proven on this, which makes **extensibility very costly** (lots to formally reprove for every new addition to the dialects).
- Is there another way? Yes…

The AtomGuard ISA: Verification Technique

- Instead of simulating the machine step-by-step, consider the **set of all valid behaviors** for each subsystem.
- We use a structure called a **sheaf**[7], which assigns to every time interval $[0,\ell]$:
  - A set of valid behaviors (enforced by predicates)
  - A "gluing" rule, which ensures that local sections can be "glued" together to form larger ones.
- The AtomGuard verifier therefore decomposes the abstract machine into sheaf subsystems.
  - Each sheaf enforces its own local constraints. No global state.
  - Adding new hardware to the model? Add another sheaf
  - Cross-subsystem constraints can be checked by combining sheaves via pullbacks (to create another sheaf!)

The AtomGuard ISA: Verification Technique

- This verification method lends itself to a **map-reduce** approach



Map stage
Check each trajectory adheres to local (sheaf) constraints in parallel

Reduce stage
Glue consistent local sections into valid global sections. Check global contracts.

Conclusion

- Neutral-Atom QPUs are a leading platform but face challenges
  - They require complex control signals (atom movements, precise zone pulses etc.)
  - Bugs in a compiled program can cause incorrect results or not be detected at all
- AtomGuard
  - A unified ISA to support any NA QPU architecture in a modular manner
  - A formal verification model for both high-level semantics and low-level hardware constraints
  - A verification technique which allows properties to be checked in parallel and extended comparably easily.

Sources

[1] Intel Itanium Schematic Image, Intel Corp. https://www.intel.com/pressroom/kits/itanium2/index.htm

[2] Quantum Computer Icon https://www.flaticon.com/free-icon/quantum-computing_10349709

[3] QuEra QPU https://thequantuminsider.com/wp-content/uploads/2024/08/Screenshot-2024-08-02-at-4.43.03-AM-1.png

[4] Pasqal QPU https://quantumzeitgeist.com/wp-content/uploads/Pasqal_QPU-2.webp

[5] Planqc QPU https://planqc.eu/media/pages/company/about-planqc/4174fdd245-1755540939/planqc-quantum-computer-in-lab-640x.jpg

[6] M. Schlosser, D. O. De Mello, D. Schäffner, T. Preuschoff, L. Kohfahl, and G. Birkl, 'Assembled arrays of Rydberg-interacting atoms', Journal of Physics B: Atomic, Molecular and Optical Physics, vol. 53, no. 14, p. 144001, 2020.

[7] A. Speranzon, D. I. Spivak, and S. Varadarajan, 'Abstraction, composition and contracts: A sheaf theoretic approach', arXiv preprint arXiv:1802.03080, 2018.

[8] T. Nipkow and G. Klein, Concrete semantics: with Isabelle/HOL. Springer, 2014.

[9] B. Coecke, 'Basic ZX-calculus for students and professionals', arXiv preprint arXiv:2303.03163, 2023.