intel foundry

Challenges in Using LLVM as a Quantum Intermediate Representation

Andrew Litteken, Albert Schmitz, Xin-Chuan Wu, Anne Matsuura

LLVM Developer's Meeting - October 24, 2024

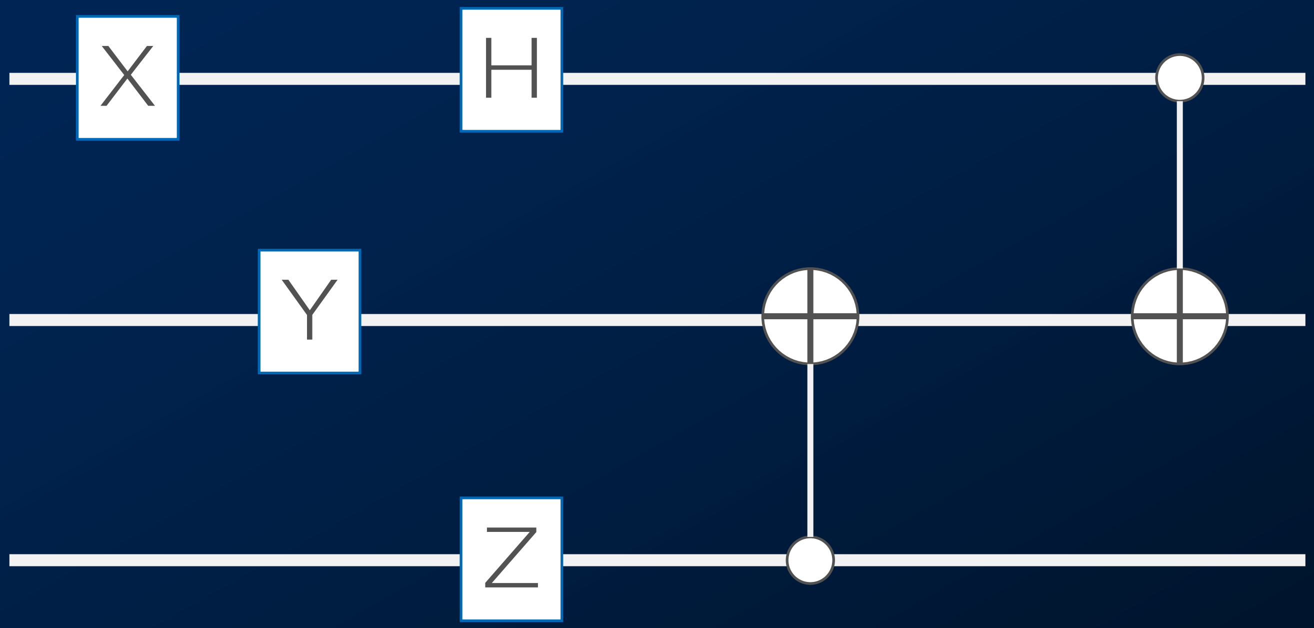
Using LLVM as a Quantum IR

- Quantum Compilation has specific constraints
- High-Level Programmers need to tools to express themselves
- Non-Compiler Users need familiar tools for Optimization

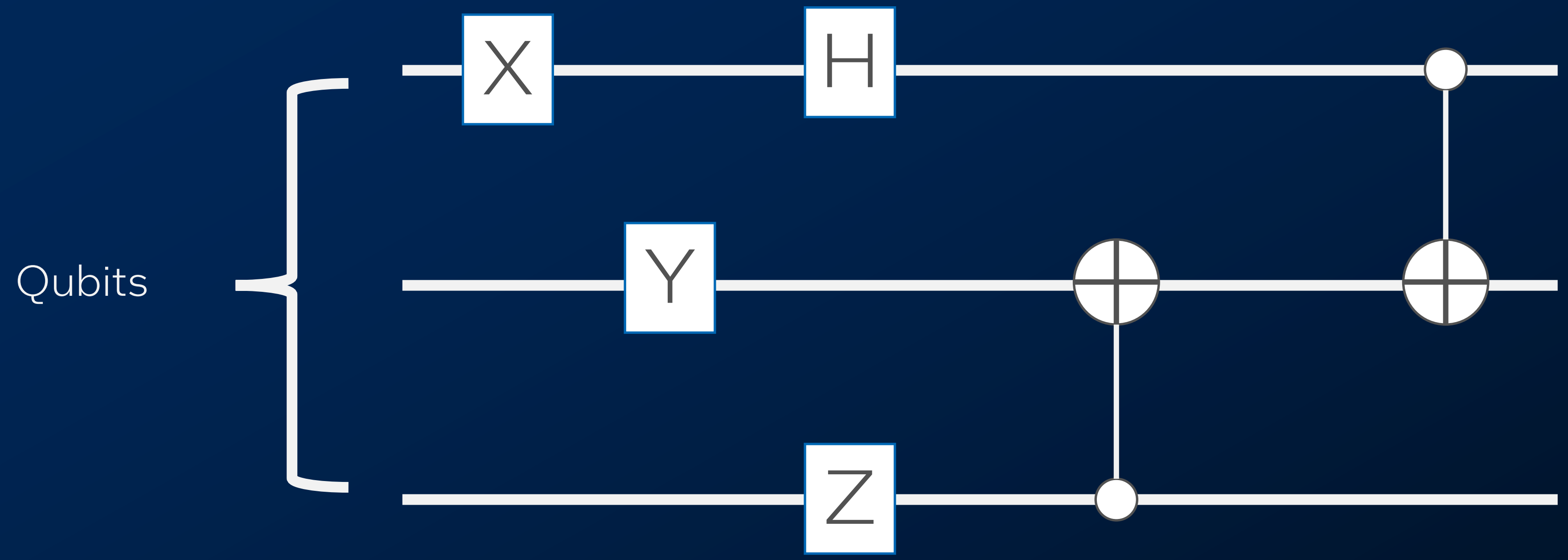
Processing Quantum Programs

intel foundry

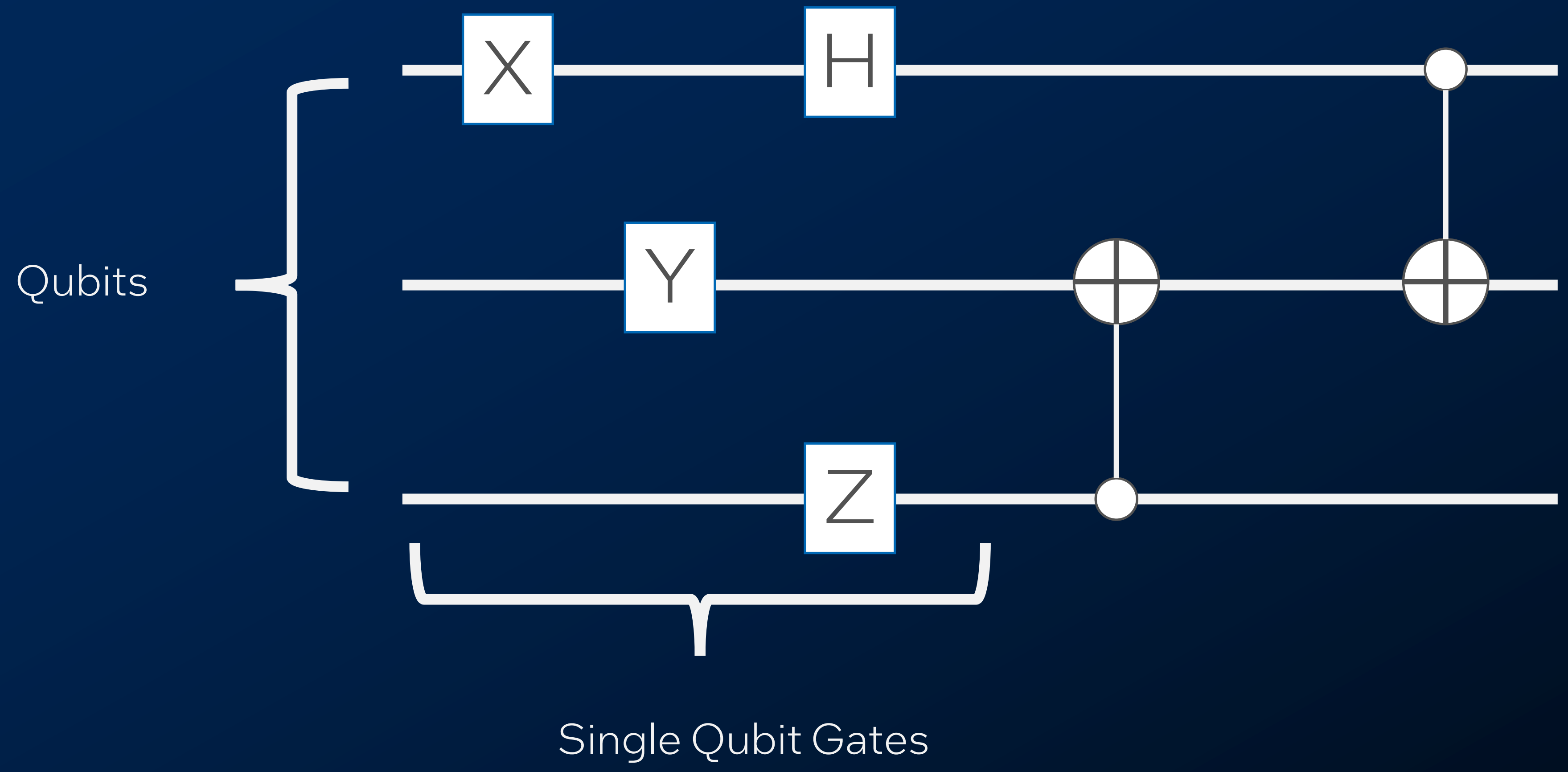
Quantum Programs



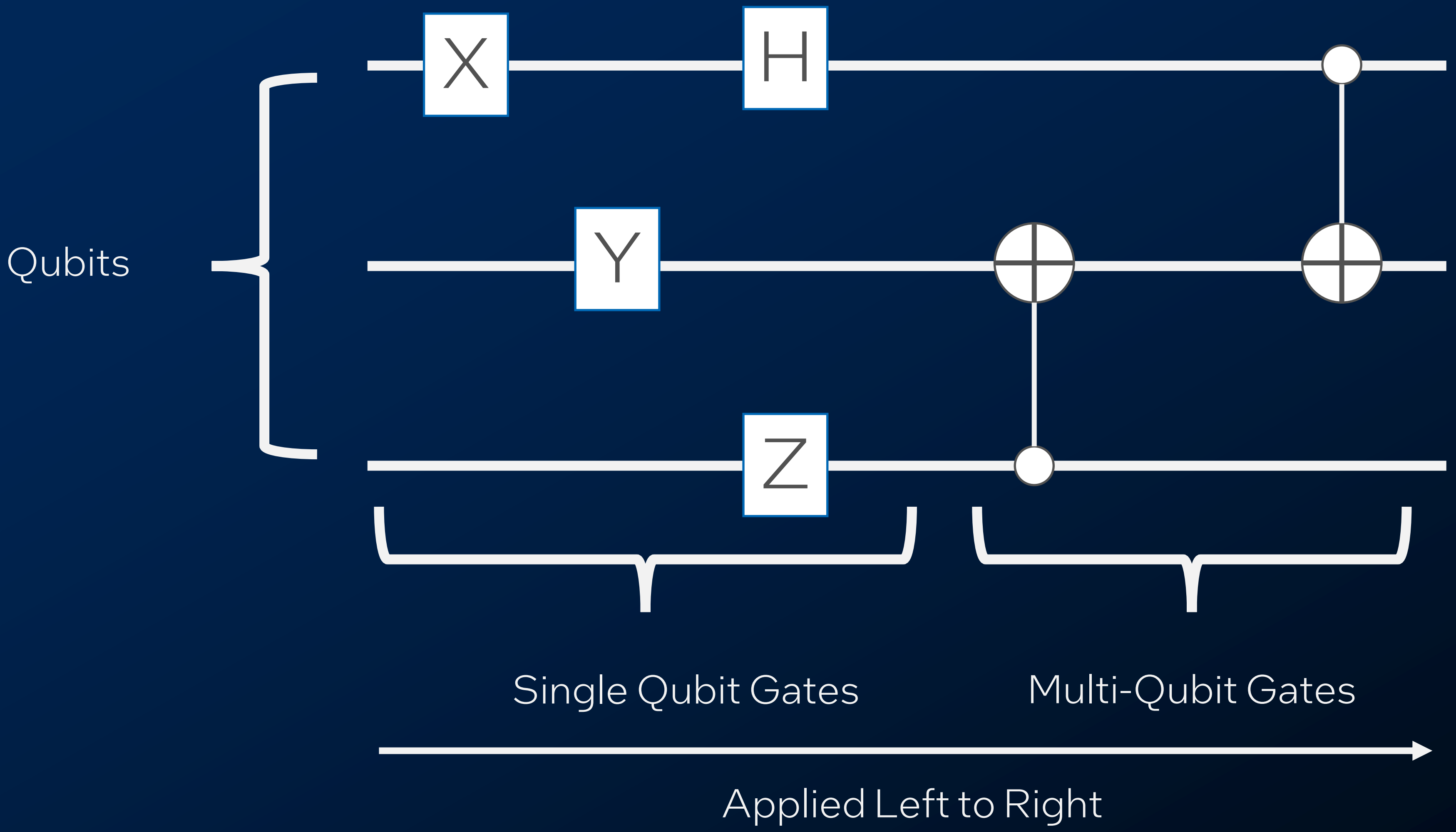
Quantum Programs



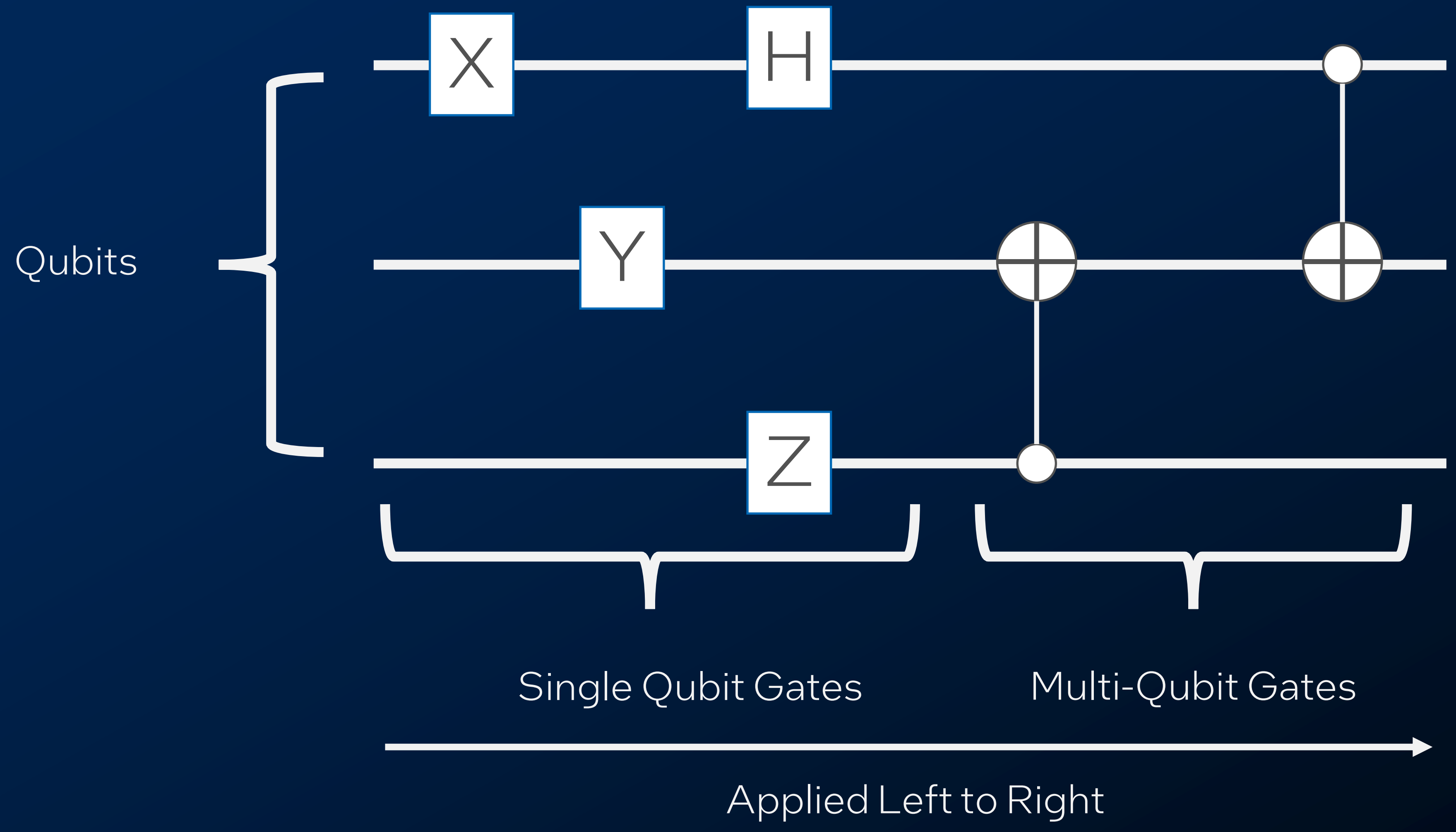
Quantum Programs



Quantum Programs

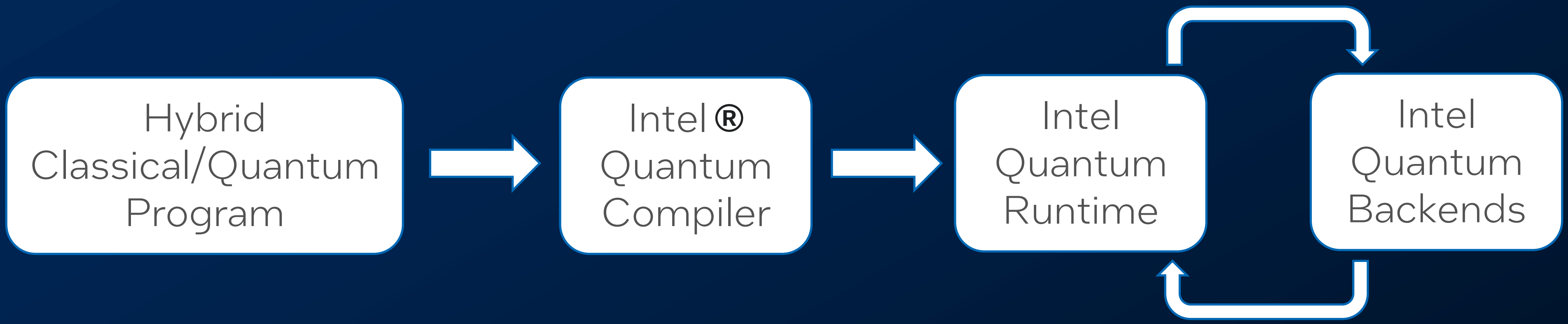


Quantum Programs



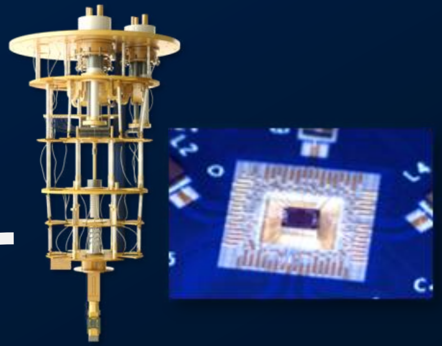
Quantum Circuit

The Intel Quantum SDK



The Intel Quantum SDK

Hybrid Classical/Quantum Program → Intel® Quantum Compiler → Intel Quantum Runtime ⇄ Intel Quantum Backends

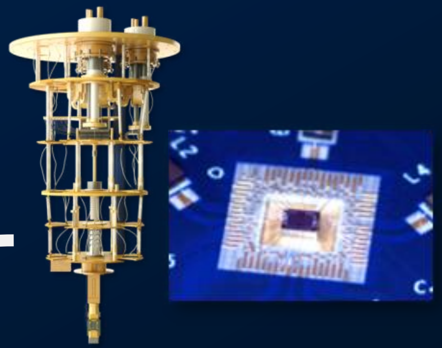


Physical Device

Simulator

The Intel Quantum SDK

Hybrid Classical/Quantum Program → Intel® Quantum Compiler → Intel Quantum Runtime ⇄ Intel Quantum Backends



Physical Device

Simulator

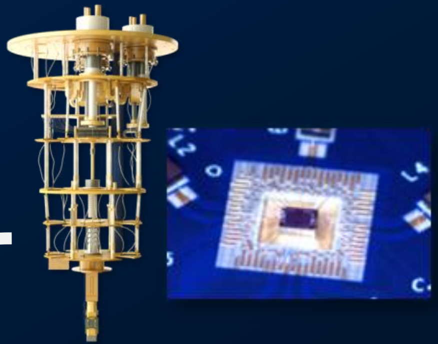
This Presentation's Focus

- Uses LLVM with Quantum Intrinsics
- Main frontend is SDK's C++ style frontend
- Could be targeted with different frontends

The Intel Quantum SDK

Hybrid Classical/Quantum Program → Intel® Quantum Compiler → Intel Quantum Runtime ⇄ Intel Quantum Backends

- Must perform classical and quantum compilation alongside each other
- Results in a module with only classical operations, and one with only quantum operations



Physical Device

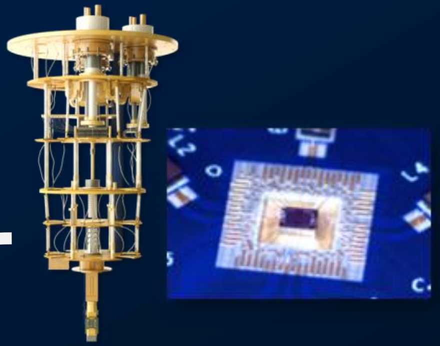
Simulator

The Intel Quantum SDK

- Deploys quantum programs to the device
- Runs classical instructions in between quantum components

Hybrid Classical/Quantum Program → Intel ® Quantum Compiler → Intel Quantum Runtime ⇄ Intel Quantum Backends

- Must perform classical and quantum compilation alongside each other
- Results in a module with only classical operations, and one with only quantum operations



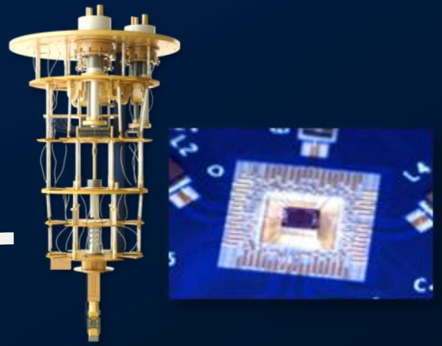
Physical Device

Simulator

The Intel Quantum SDK

- Deploys quantum programs to the device
- Runs classical instructions in between quantum components

Hybrid Classical/Quantum Program → Intel ® Quantum Compiler → Intel Quantum Runtime ⇄ Intel Quantum Backends



Physical Device

Simulator

- Must perform classical and quantum compilation alongside each other
- Results in a module with only classical operations, and one with only quantum operations

- Uses runtime instructions for execution
- Currently cannot perform conditional execution
- Returns execution information to the run time
- Limited resources

The Compiler has many Quantum Challenges

- No Branching
- Match Device Constraints
- Reduce Quantum Operations
- No Classical Operations in Quantum Code

The Compiler has many Quantum Challenges

- No Branching
- Match Device Constraints
- Reduce Quantum Operations
- No Classical Operations in Quantum Code

These are constraints based on today’s technology

Handling Constraints with LLVM

intel foundry

Quantum Primitives for Hybrid Programs

Quantum Kernel
(Can be nested)

```
#include <clang/Quantum/quintrinsics.h>

/// Quantum Runtime Library APIs
#include <quantum_full_state_simulator_backend.h>

const int total_qubits = 2;
qbit qubit_register[total_qubits];

quantum_kernel void ghz_total_qubits() {
  for (int i = 0; i < total_qubits; i++) {
    PrepZ(qubit_register[i]);
  }

  H(qubit_register[0]);

  for (int i = 0; i < total_qubits - 1; i++) {
    CNOT(qubit_register[i], qubit_register[i + 1]);
  }
}
```

Quantum Primitives for Hybrid Programs

Quantum Kernel
(Can be nested)

```
#include <clang/Quantum/quintrinsics.h>

/// Quantum Runtime Library APIs
#include <quantum_full_state_simulator_backend.h>

const int total_qubits = 2;
qbit qubit_register[total_qubits];

quantum_kernel void ghz_total_qubits() {
  for (int i = 0; i < total_qubits; i++) {
    PrepZ(qubit_register[i]);
  }

  H(qubit_register[0]);

  for (int i = 0; i < total_qubits - 1; i++) {
    CNOT(qubit_register[i], qubit_register[i + 1]);
  }
}
```

Kernel Annotation

- `quantum_kernel` adds section attribute

Quantum Primitives for Hybrid Programs

Quantum Type

Quantum Kernel
(Can be nested)

Kernel Annotation

```
#include <clang/Quantum/quintrinsics.h>

/// Quantum Runtime Library APIs
#include <quantum_full_state_simulator_backend.h>

const int total_qubits = 2;
qbit qubit_register[total_qubits];

quantum_kernel void ghz_total_qubits() {
  for (int i = 0; i < total_qubits; i++) {
    PrepZ(qubit_register[i]);
  }

  H(qubit_register[0]);

  for (int i = 0; i < total_qubits - 1; i++) {
    CNOT(qubit_register[i], qubit_register[i + 1]);
  }
}
```

- `quantum_kernel` adds section attribute
- `qbit` and `cbit` are typedefs over particular integer types

Quantum Primitives for Hybrid Programs

Quantum Type

Quantum Kernel (Can be nested)

```
#include <clang/Quantum/quintrinsics.h>

/// Quantum Runtime Library APIs
#include <quantum_full_state_simulator_backend.h>

const int total_qubits = 2;
qbit qubit_register[total_qubits];

quantum_kernel void ghz_total_qubits() {
  for (int i = 0; i < total_qubits; i++) {
    PrepZ(qubit_register[i]);
  }

  H(qubit_register[0]);

  for (int i = 0; i < total_qubits - 1; i++) {
    CNOT(qubit_register[i], qubit_register[i + 1]);
  }
}
```

Kernel Annotation

Quantum Operations

- **`quantum_kernel`** adds section attribute
- **`qbit`** and **`cbit`** are typedefs over particular integer types
- **`quintrinsics.h`** defines quantum functions

Users Can Write Code that Mixes Classical And Quantum Computation

Quantum Type

Quantum Kernel
(Can be nested)

Kernel Annotation

```
#include <clang/Quantum/quintrinsics.h>

/// Quantum Runtime Library APIs
#include <quantum_full_state_simulator_backend.h>

const int total_qubits = 2;
qbit qubit_register[total_qubits];

quantum_kernel void ghz_total_qubits() {
  for (int i = 0; i < total_qubits; i++) {
    PrepZ(qubit_register[i]);
  }

  H(qubit_register[0]);

  for (int i = 0; i < total_qubits - 1; i++) {
    CNOT(qubit_register[i], qubit_register[i + 1]);
  }
}
```

Quantum Operations

Classical Component
(calls quantum components)

```
int main() {
  iqsdk::DeviceConfig qd_config("QD_SIM");
  iqsdk::FullStateSimulator quantum_8086(qd_config);
  if (iqsdk::QRT_ERROR_SUCCESS != quantum_8086.ready())
    return 1;

  // get references to qbits
  std::vector<std::reference_wrapper<qbit>> qids;
  for (int id = 0; id < total_qubits; ++id) {
    qids.push_back(std::ref(qubit_register[id]));
  }

  ghz_total_qubits();

  iqsdk::QssMap<double> probability_map;
  probability_map = quantum_8086.getProbabilities(qids, bases);
}
```

Users Can Write Code that Mixes Classical And Quantum Computation

Quantum Type

Quantum Kernel
(Can be nested)

Kernel Annotation

Quantum Operations

```cpp
#include <clang/Quantum/quintrinsics.h>

/// Quantum Runtime Library APIs
#include <quantum_full_state_simulator_backend.h>

const int total_qubits = 2;
qbit qubit_register[total_qubits];

quantum_kernel void ghz_total_qubits() {
  for (int i = 0; i < total_qubits; i++) {
    PrepZ(qubit_register[i]);
  }

  H(qubit_register[0]);

  for (int i = 0; i < total_qubits - 1; i++) {
    CNOT(qubit_register[i], qubit_register[i + 1]);
  }
}
```

Classical Component
(calls quantum components)

```cpp
int main() {
  iqsdk::DeviceConfig qd_config("QD_SIM");
  iqsdk::FullStateSimulator quantum_8086(qd_config);
  if (iqsdk::QRT_ERROR_SUCCESS != quantum_8086.ready())
    return 1;

  // get references to qbits
  std::vector<std::reference_wrapper<qbit>> qids;
  for (int id = 0; id < total_qubits; ++id) {
    qids.push_back(std::ref(qubit_register[id]));
  }

  ghz_total_qubits();

  iqsdk::QssMap<double> probability_map;
  probability_map = quantum_8086.getProbabilities(qids, bases);
}
```

LLVM Provides a Leg Up to Compilation

Quantum Kernel
(Can be nested)

```
#include <clang/Quantum/quintrinsics.h>

/// Quantum Runtime Library APIs
#include <quantum_full_state_simulator_backend.h>

const int total_qubits = 2;
qbit qubit_register[total_qubits];

quantum_kernel void ghz_total_qubits() {
  for (int i = 0; i < total_qubits; i++) {
    PrepZ(qubit_register[i]);
  }

  H(qubit_register[0]);

  for (int i = 0; i < total_qubits - 1; i++) {
    CNOT(qubit_register[i], qubit_register[i + 1]);
  }
}
```

Classical Component
(calls quantum components)

```
int main() {
  iqsdk::DeviceConfig qd_config("QD_SIM");
  iqsdk::FullStateSimulator quantum_8086(qd_config);
  if (iqsdk::QRT_ERROR_SUCCESS != quantum_8086.ready())
    return 1;

  // get references to qbits
  std::vector<std::reference_wrapper<qbit>> qids;
  for (int id = 0; id < total_qubits; ++id) {
    qids.push_back(std::ref(qubit_register[id]));
  }

  ghz_total_qubits();

  iqsdk::QssMap<double> probability_map;
  probability_map = quantum_8086.getProbabilities(qids, bases);
}
```

Conditional Structures
with Quantum Operations

Structures are Found in Classical and Quantum

Quantum Kernel
(Can be nested)

```
#include <clang/Quantum/quintrinsics.h>

/// Quantum Runtime Library APIs
#include <quantum_full_state_simulator_backend.h>

const int total_qubits = 2;
qbit qubit_register[total_qubits];

quantum_kernel void ghz_total_qubits() {
  for (int i = 0; i < total_qubits; i++) {
    PrepZ(qubit_register[i]);
  }

  H(qubit_register[0]);

  for (int i = 0; i < total_qubits - 1; i++) {
    CNOT(qubit_register[i], qubit_register[i + 1]);
  }
}
```

Classical Component
(calls quantum components)

```
int main() {
  iqsdk::DeviceConfig qd_config("QD_SIM");
  iqsdk::FullStateSimulator quantum_8086(qd_config);
  if (iqsdk::QRT_ERROR_SUCCESS != quantum_8086.ready())
    return 1;

  // get references to qbits
  std::vector<std::reference_wrapper<qbit>> qids;
  for (int id = 0; id < total_qubits; ++id) {
    qids.push_back(std::ref(qubit_register[id]));
  }

  ghz_total_qubits();

  iqsdk::QssMap<double> probability_map;
  probability_map = quantum_8086.getProbabilities(qids, bases);
}
```

Conditional Structures
with Quantum Operations and classical

Pass Managers Provide Flexibility

Inlining Kernels and Adding Intrinsics

Pre-Unrolling Preparation

Loop Unrolling and Constant Folding

Post Unrolling Cleanup

Quantum Handling

Pass Managers Provide Flexibility

Inlining Kernels and Adding Intrinsics

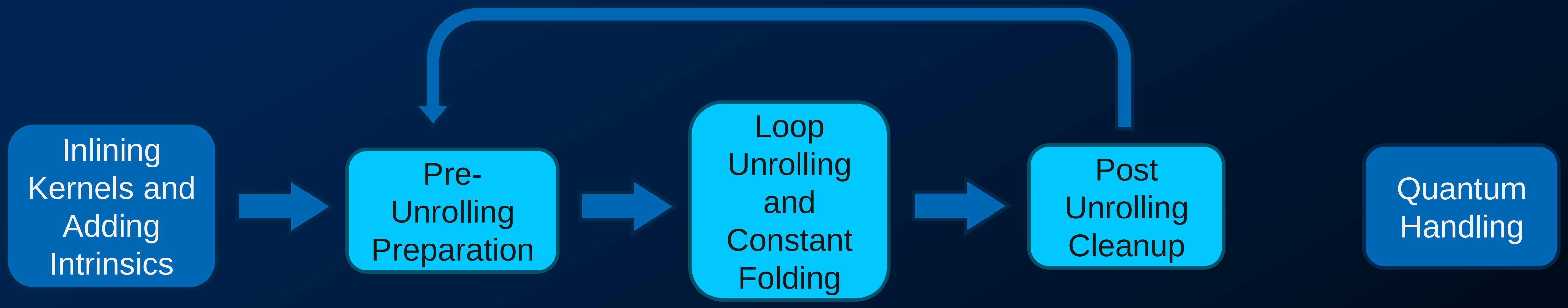
Pre-Unrolling Preparation

Loop Unrolling and Constant Folding

Post Unrolling Cleanup

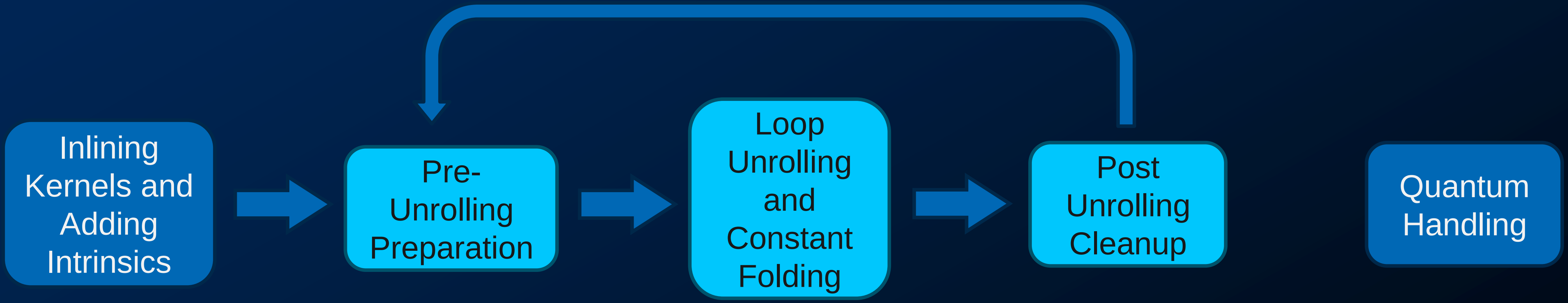
Quantum Handling

Pass Managers Provide Flexibility



Pass Managers Provide Flexibility

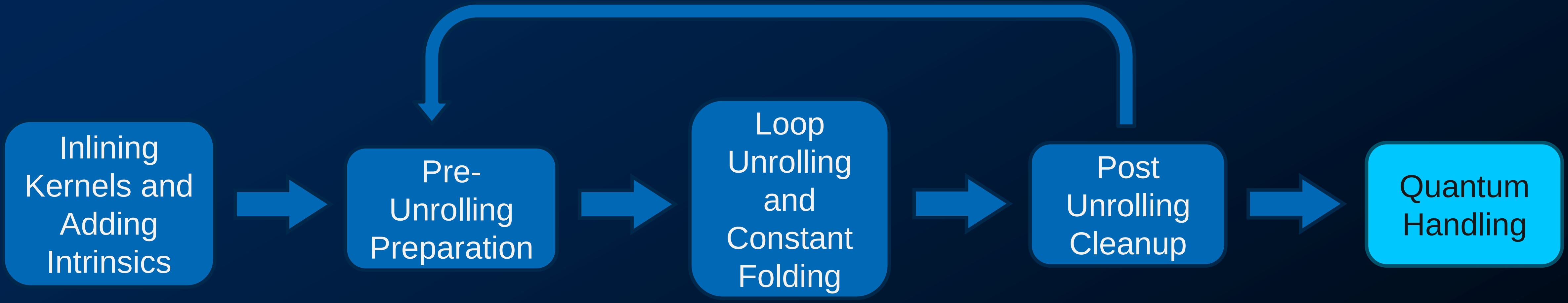
Re-runs until all marked kernels valid, or attempt number exceeded.



- Only runs on marked quantum kernels
- Only runs Loop optimizations on loops that contain quantum operations

Pass Managers Provide Flexibility

Re-runs until all marked kernels valid, or attempt number exceeded.



- Only runs on marked quantum kernels
- Only runs Loop optimizations on loops that contain quantum operations

Quantum Optimizations Need Quantum Types in IR

Quantum Optimizations Need Quantum Types in IR

Declaration of Quantum Operation in
Intermediate Representation

```
define dso_local void @_Z1HRt(ptr noundef nonnull align 2 dereferenceable(2) %q) #0 {
entry:
  call void @llvm.quantum.qubit(ptr noundef nonnull %q)
  ret void
}
```

Quantum Optimizations Need Quantum Types in IR

Declaration of Quantum Operation in Intermediate Representation

```
define dso_local void @_Z1HRt(ptr noundef nonnull align 2 dereferenceable(2) %q) #0 {
entry:
  call void @llvm.quantum.qubit(ptr noundef nonnull %q)
  ret void
}
```

Mapping Qubit to an Argument

Quantum Optimizations Need Quantum Types in IR

Declaration of Quantum Operation in Intermediate Representation

```
define dso_local void @_Z1HRt(ptr noundef nonnull align 2 dereferenceable(2) %q) #0 {
entry:
  call void @llvm.quantum.qubit(ptr noundef nonnull %q)
  ret void
}
```

Marking as a qubit reference

```
define dso_local void @_Z1HRt(ptr noundef nonnull align 2 dereferenceable(2) "qubit_ref" %q) #0 {
entry:
  call void @llvm.quantum.qubit(ptr noundef nonnull %q)
  ret void
}
```

Providing Tools to Users

Writing Code Within these Constraints is Hard

Standard SDK

```
quantum_kernel void qft() {
  // qft non-recursive implementation

  // Apply H and rotations
  // Starting from qubit 0
  for (int index = 0; index < N; index++) {
    H(QubitReg[index]);
    for (int index_r = 1; index_r < N - index; index_r++) {
      double angle = 2 * (1 / M_1_PI) / std::pow(2, index_r
+ 1);
      CPhase(QubitReg[index + index_r], QubitReg[index],
angle);
    }
  }

  // Add SWAP gates
  for (int q_index = 0; q_index < std::floor(N / 2);
q_index++) {
    SWAP(QubitReg[q_index], QubitReg[N - q_index - 1]);
  }
}
```

Functional Language Extension for Quantum (FLEQ) Makes it Easier

Standard SDK

```
quantum_kernel void qft() {
  // qft non-recursive implementation

  // Apply H and rotations
  // Starting from qubit 0
  for (int index = 0; index < N; index++) {
    H(QubitReg[index]);
    for (int index_r = 1; index_r < N - index; index_r++) {
      double angle = 2 * (1 / M_1_PI) / std::pow(2, index_r
+ 1);
      CPhase(QubitReg[index + index_r], QubitReg[index],
angle);
    }
  }

  // Add SWAP gates
  for (int q_index = 0; q_index < std::floor(N / 2);
q_index++) {
    SWAP(QubitReg[q_index], QubitReg[N - q_index - 1]);
  }
}
```

FLEQ

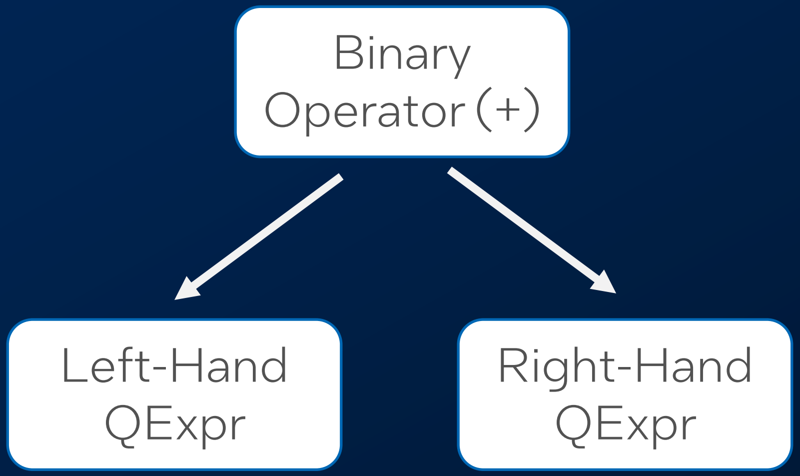
```
PROTECT QExpr qftCPhaseLadder(qbit& q, qlist::QList
reg){
  int sz = reg.size();
  double theta = - M_PI / pow(2, sz);

  return qexpr::cIfTrue(sz > 0,
              qexpr::_CPhase(q, reg[0], theta)
              + qftCPhaseLadder(q, reg+1));
}
PROTECT QExpr qftHelper(qlist::QList reg){
  int sz = reg.size();
  return qexpr::cIfTrue(sz > 0,
      qexpr::_H(reg[sz-1]) +
        qftCPhaseLadder(reg[sz-1],
                        reg<<1)
         + qftHelper(reg<<1));
}

QExpr qft(qlist::QList reg){
  return qftHelper(reg) + reverseRegister(reg);
}
```

Clang Plugins Enable FLEQ

```
PROTECT QExpr ghz(qlist::QList qs){
  int len = qs.size();
  return (
    qexpr::map(qexpr::_PrepZ, qs)
    + qexpr::_H(qs[0])
    + qexpr::map(qexpr::_CNOT,
         qs(0,len-1), qs(1,len)));
}
```

Clang Plugins Enable FLEQ

```
PROTECT QExpr ghz(qlist::QList qs){
  int len = qs.size();
  return (
    qexpr::map(qexpr::_PrepZ, qs)
    + qexpr::_H(qs[0])
    + qexpr::map(qexpr::_CNOT,
         qs(0,len-1), qs(1,len)));
}
```

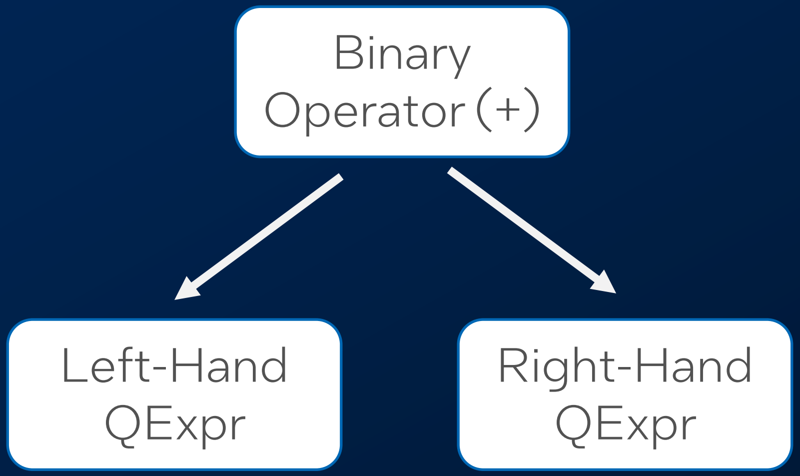


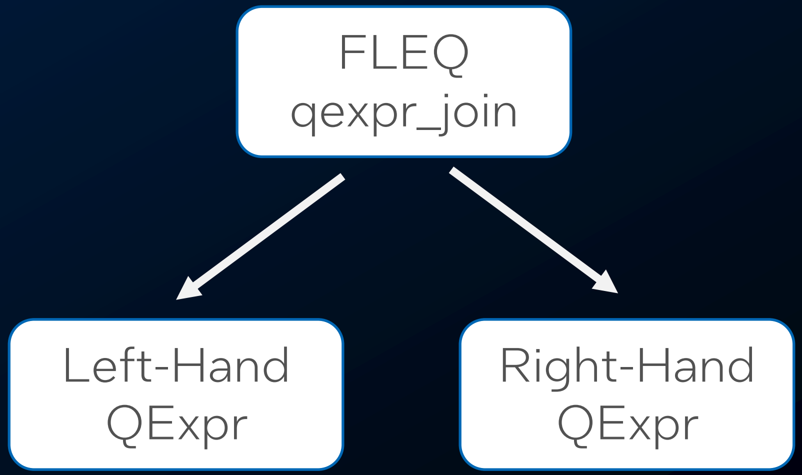
Clang Plugins Enable FLEQ

```
PROTECT QExpr ghz(qlist::QList qs){
  int len = qs.size();
  return (
    qexpr::map(qexpr::_PrepZ, qs)
    + qexpr::_H(qs[0])
    + qexpr::map(qexpr::_CNOT,
         qs(0,len-1), qs(1,len)));
}
```

→ FLEQ Clang Plugin →

```
PROTECT QExpr ghz(qlist::QList qs){
  int len = qs.size();
  return (
    qexpr::join(
      qexpr::map(
        qexpr::_PrepZ, qs),
      qexpr::join(
        qexpr::_H(qs[0]),
        qexpr::map(
         qexpr::_CNOT,
         qs(0,len-1), qs(1,len))
       )
     ));
}
```





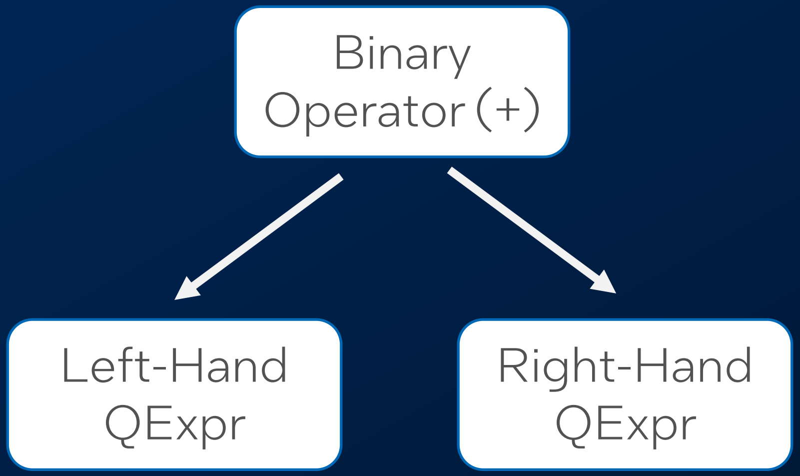
Clang Plugins Enable FLEQ

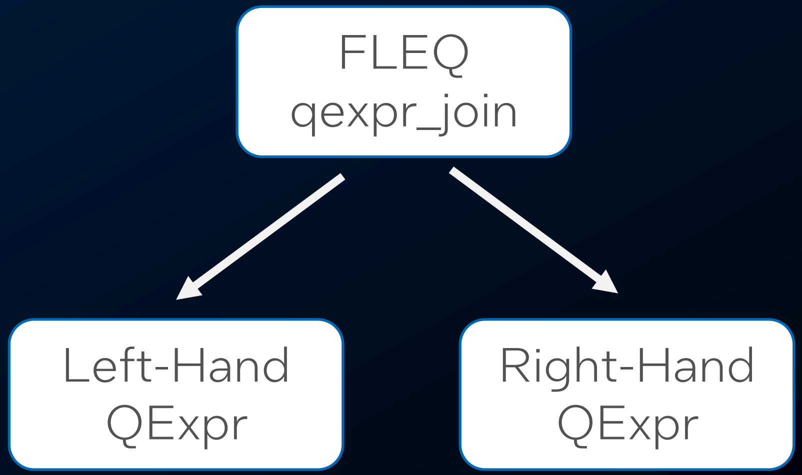
```
PROTECT QExpr ghz(qlist::QList qs){
  int len = qs.size();
  return (
    qexpr::map(qexpr::_PrepZ, qs)
    + qexpr::_H(qs[0])
    + qexpr::map(qexpr::_CNOT,
         qs(0,len-1), qs(1,len)));
}
```


```
PROTECT QExpr ghz(qlist::QList qs){
  int len = qs.size();
  return (
    qexpr::join(
      qexpr::map(
        qexpr::_PrepZ, qs),
      qexpr::join(
        qexpr::_H(qs[0]),
        qexpr::map(
         qexpr::_CNOT,
         qs(0,len-1), qs(1,len))
        )
      ));
}
```

Replaced Operands

- Binary Add (join)
- Binary Multiply (join)
- ~, !, - (invert)
- ^ (power)
- << (bind)
- >> (bind)





FLEQ Enables More Advanced Optimization

FLEQ Enables More Advanced Optimization

FLEQ Modified IR Module → Internal Graph Representation → Graph Based Quantum Optimizations → IR Module with Reduced Quantum Resources

Original Hybrid Program

```
quantum_kernel void qft() {
  . . .
}

quantum_kernel void qft_inverse() {
  . . .
}

quantum_kernel void qft_all() {
  prepZAll();
  qft();
  qft_inverse();
}
```

Explicit Inverse

FLEQ with inferred inversion

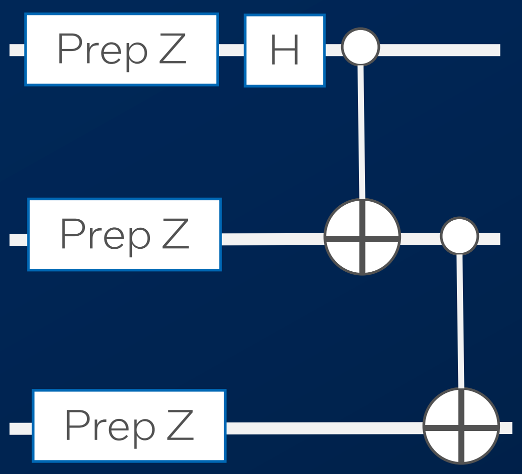
```
QExpr qft(qlist::QList reg){
  return qftHelper(reg) + reverseRegister(reg);
}

int main() {
  qexpr::eval_hold(fourierBasis(qs, compBasisIndex)
                    + -qft(qs));
}
```

Implicit Inverse

Simplifying Optimization Development

intel foundry

LLVM Doesn't Match Quantum Abstractions



Circuit Form

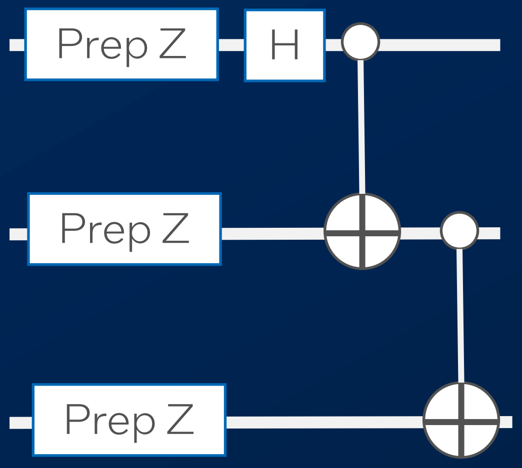
```
quantum_kernel void ghz_total_qubits() {
  for (int i = 0; i < total_qubits; i++) {
    PrepZ(q[i]);
  }

  H(q[0]);

  for (int i = 0; i < total_qubits - 1;
i++) {
    CNOT(q[i], q[i + 1]);
  }
}
```

SDK Form

```
aqcc.quantum:
  %arrayidx34 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 0
  call void @_Z5PrepZRt(ptr %arrayidx34)
  %arrayidx33 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 1
  call void @_Z5PrepZRt(ptr %arrayidx33)
  %arrayidx32 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 2
  call void @_Z1HRt(ptr %arrayidx22)
  %arrayidx20 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 0
  %arrayidx21 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 1
  call void @_Z4CNOTRtS_(ptr %arrayidx20,
ptr %arrayidx21)
  %arrayidx18 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 1
  %arrayidx19 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 2
  call void @_Z4CNOTRtS_(ptr %arrayidx18,
ptr %arrayidx19)
  br label %aqcc.meas.move.end
```

IR Form (Post Unrolling)

LLVM Doesn’t Match Quantum Abstractions



Circuit Form

Typical Abstraction for Optimization

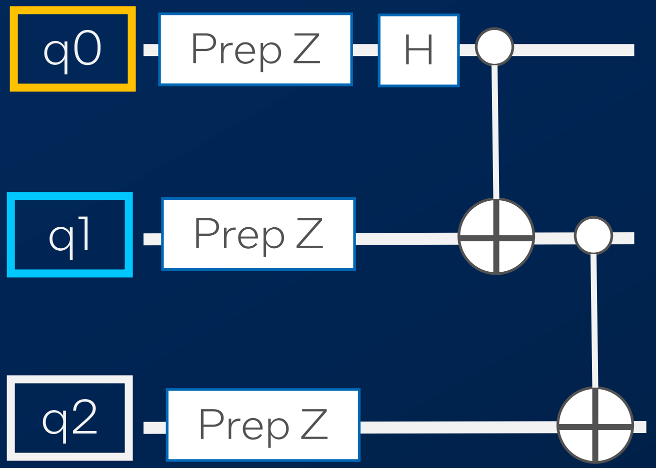
```
quantum_kernel void ghz_total_qubits() {
  for (int i = 0; i < total_qubits; i++) {
    PrepZ(q[i]);
  }

  H(q[0]);

  for (int i = 0; i < total_qubits - 1;
i++) {
    CNOT(q[i], q[i + 1]);
  }
}
```

SDK Form

```
aqcc.quantum:
  %arrayidx34 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 0
  call void @_Z5PrepZRt(ptr %arrayidx34)
  %arrayidx33 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 1
  call void @_Z5PrepZRt(ptr %arrayidx33)
  %arrayidx32 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 2
  call void @_Z1HRt(ptr %arrayidx22)
  %arrayidx20 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 0
  %arrayidx21 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 1
  call void @_Z4CNOTRtS_(ptr %arrayidx20,
ptr %arrayidx21)
  %arrayidx18 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 1
  %arrayidx19 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 2
  call void @_Z4CNOTRtS_(ptr %arrayidx18,
ptr %arrayidx19)
  br label %aqcc.meas.move.end
```

IR Form (Post Unrolling)

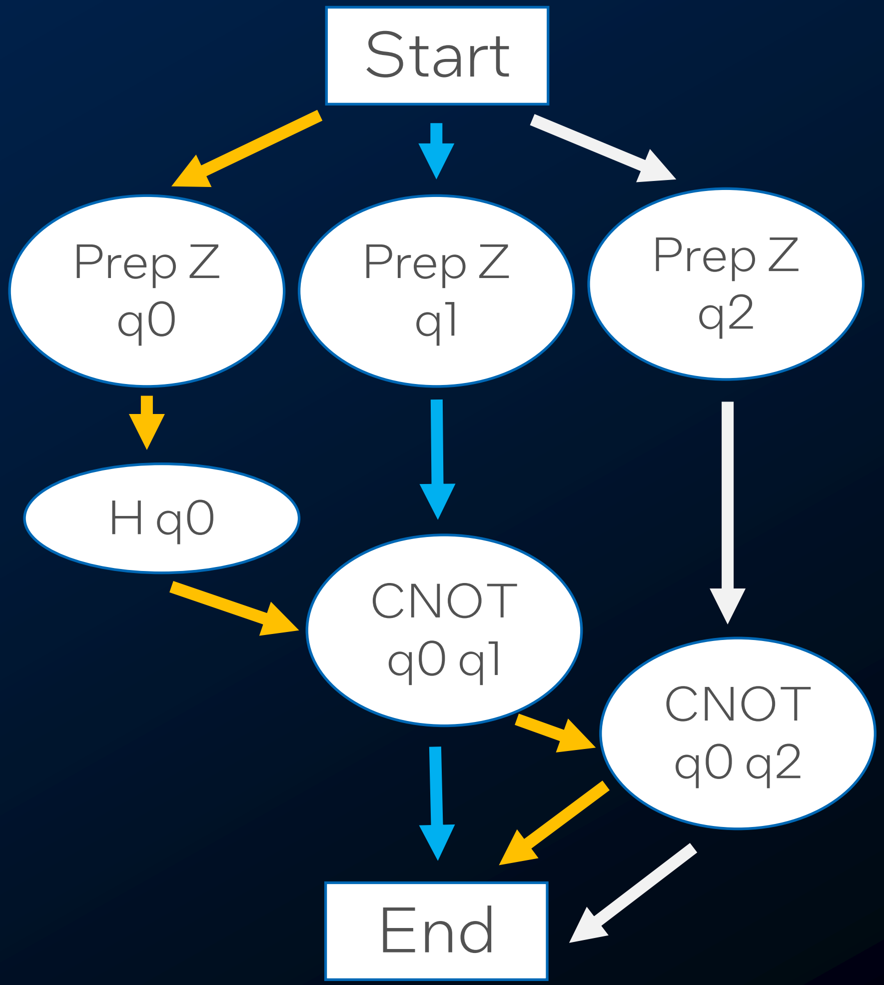
The Quantum Circuit Object is an LLVM Wrapper



Circuit Form

```
aqcc.quantum:
  %arrayidx34 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 0
  call void @_Z5PrepZRt(ptr %arrayidx34)
  %arrayidx33 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 1
  call void @_Z5PrepZRt(ptr %arrayidx33)
  %arrayidx32 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 2
  call void @_Z1HRt(ptr %arrayidx22)
  %arrayidx20 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 0
  %arrayidx21 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 1
  call void @_Z4CNOTRtS_(ptr %arrayidx20,
ptr %arrayidx21)
  %arrayidx18 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 1
  %arrayidx19 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 2
  call void @_Z4CNOTRtS_(ptr %arrayidx18,
ptr %arrayidx19)
  br label %aqcc.meas.move.end
```

IR Form (Post Unrolling)

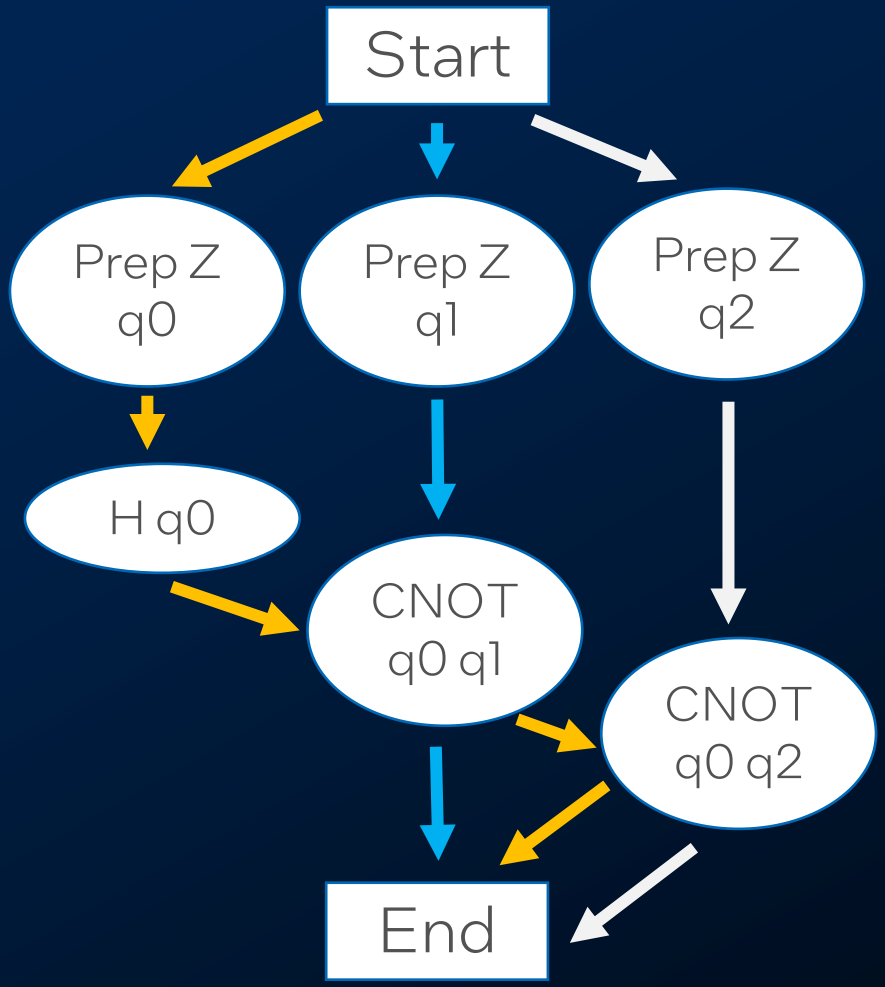


Quantum Circuit Object

Circuit Object Manipulation is Reflected in IR

```
aqcc.quantum:
  . . .
  %arrayidx33 = getelementptr inbounds [12
x i16], ptr @Qumem, i64 0, i64 1
  . . .
  %arrayidx20 = getelementptr inbounds [12
x i16], ptr @Qumem, i64 0, i64 0
  %arrayidx21 = getelementptr inbounds [12
x i16], ptr @Qumem, i64 0, i64 1
  call void @_Z4CNOTRtS_(ptr %arrayidx20,
ptr %arrayidx21)
  . . .
```

IR Form



Quantum Circuit Object

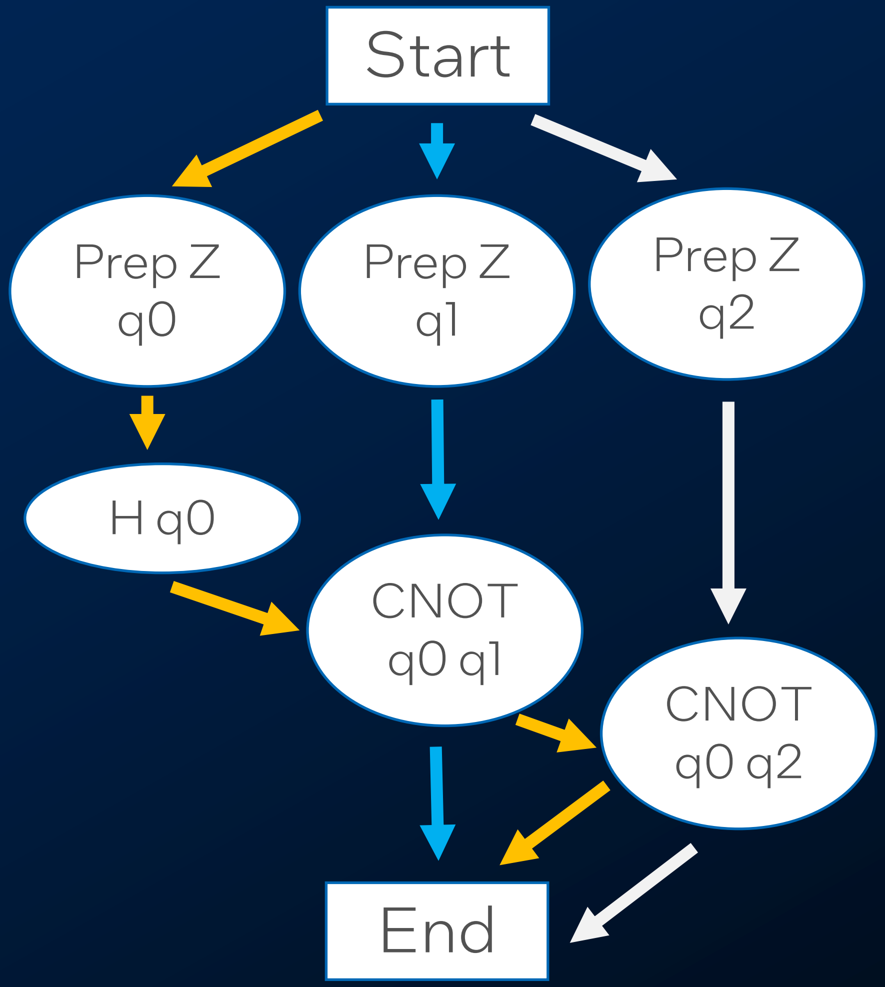
Supported Options

- Deletion
- Insertion
- Moving Operations
- New Operations
- New Qubits
- Consistent Iteration

Circuit Object Manipulation is Reflected in IR

```
aqcc.quantum:
  . . .
  %arrayidx33 = getelementptr inbounds [12
x i16], ptr @Qumem, i64 0, i64 1
  . . .
  %arrayidx20 = getelementptr inbounds [12
x i16], ptr @Qumem, i64 0, i64 0
  %arrayidx21 = getelementptr inbounds [12
x i16], ptr @Qumem, i64 0, i64 1
  call void @_Z4CNOTRtS_(ptr %arrayidx20,
ptr %arrayidx21)
  . . .
```

IR Form



Quantum Circuit Object

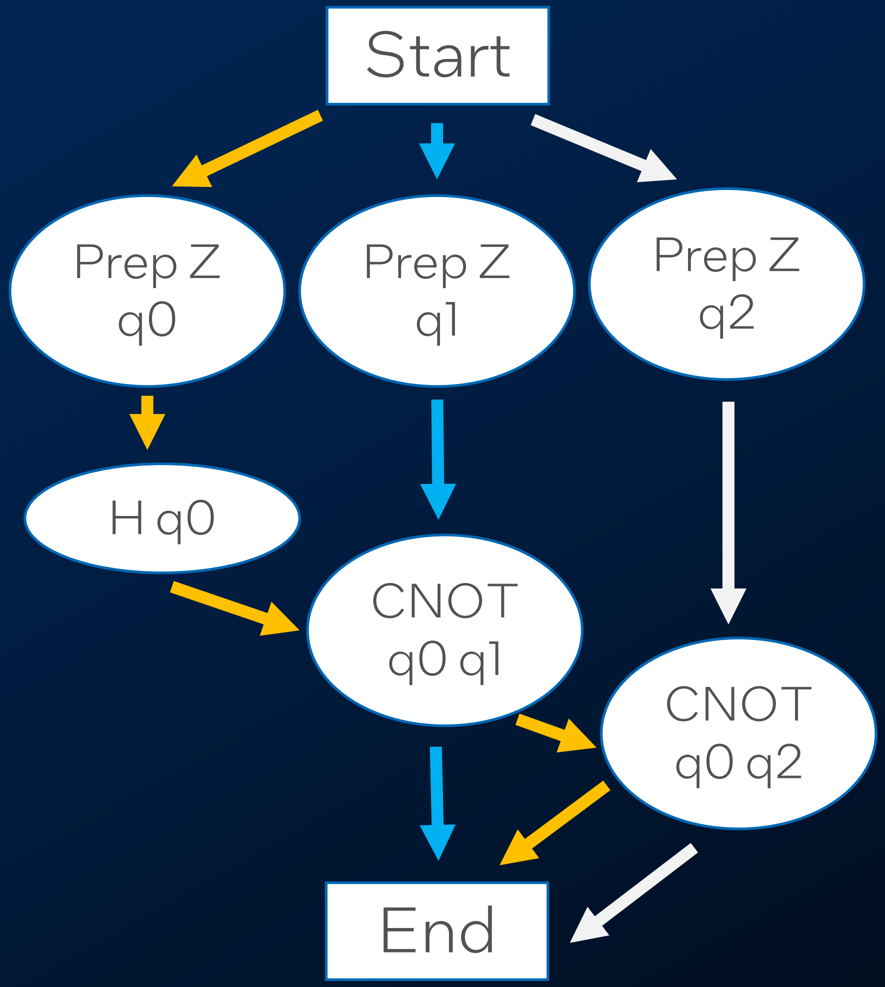
Supported Options

- Deletion
- Insertion
- Moving Operations
- New Operations
- New Qubits
- Consistent Iteration

Circuit Object Manipulation is Reflected in IR

```
aqcc.quantum:
  . . .
  %arrayidx33 = getelementptr inbounds [12
x i16], ptr @Qumem, i64 0, i64 1
  . . .
  %arrayidx20 = getelementptr inbounds [12
x i16], ptr @Qumem, i64 0, i64 0
  %arrayidx21 = getelementptr inbounds [12
x i16], ptr @Qumem, i64 0, i64 1
  call void @_Z4CNOTRtS_(ptr %arrayidx20,
ptr %arrayidx21)
  . . .
```

IR Form



Quantum Circuit Object

Supported Options

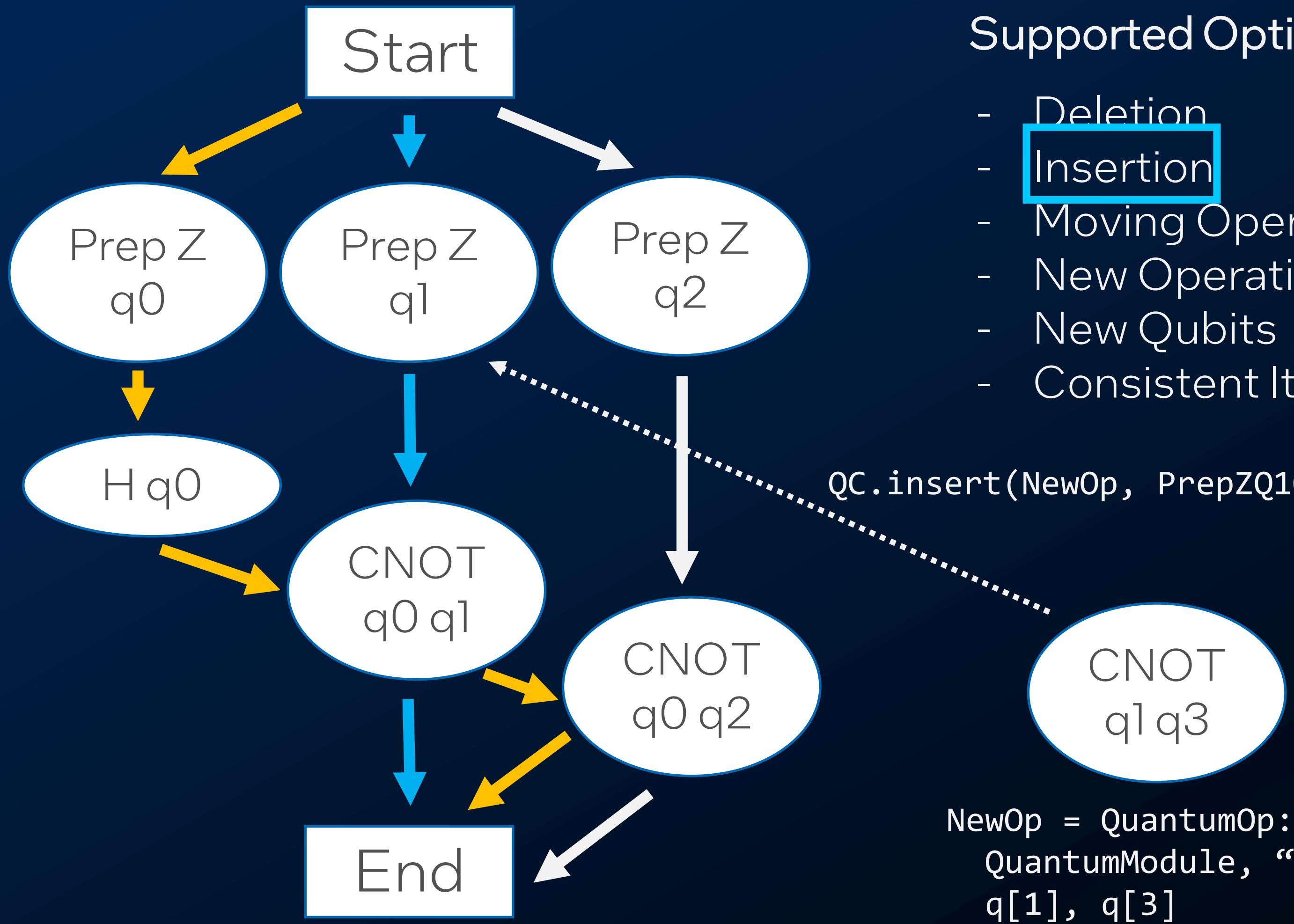
- Deletion
- Insertion
- Moving Operations
- New Operations
- New Qubits
- Consistent Iteration


```
NewOp = QuantumOp::Create(
  QuantumModule, “cnot”,
  q[1], q[3]
)
```

Circuit Object Manipulation is Reflected in IR

```
aqcc.quantum:
  . . .
  %arrayidx33 = getelementptr inbounds [12
x i16], ptr @Qumem, i64 0, i64 1
  . . .
  %arrayidx20 = getelementptr inbounds [12
x i16], ptr @Qumem, i64 0, i64 0
  %arrayidx21 = getelementptr inbounds [12
x i16], ptr @Qumem, i64 0, i64 1
  call void @_Z4CNOTRtS_(ptr %arrayidx20,
ptr %arrayidx21)
  . . .
```

IR Form



Quantum Circuit Object

Supported Options

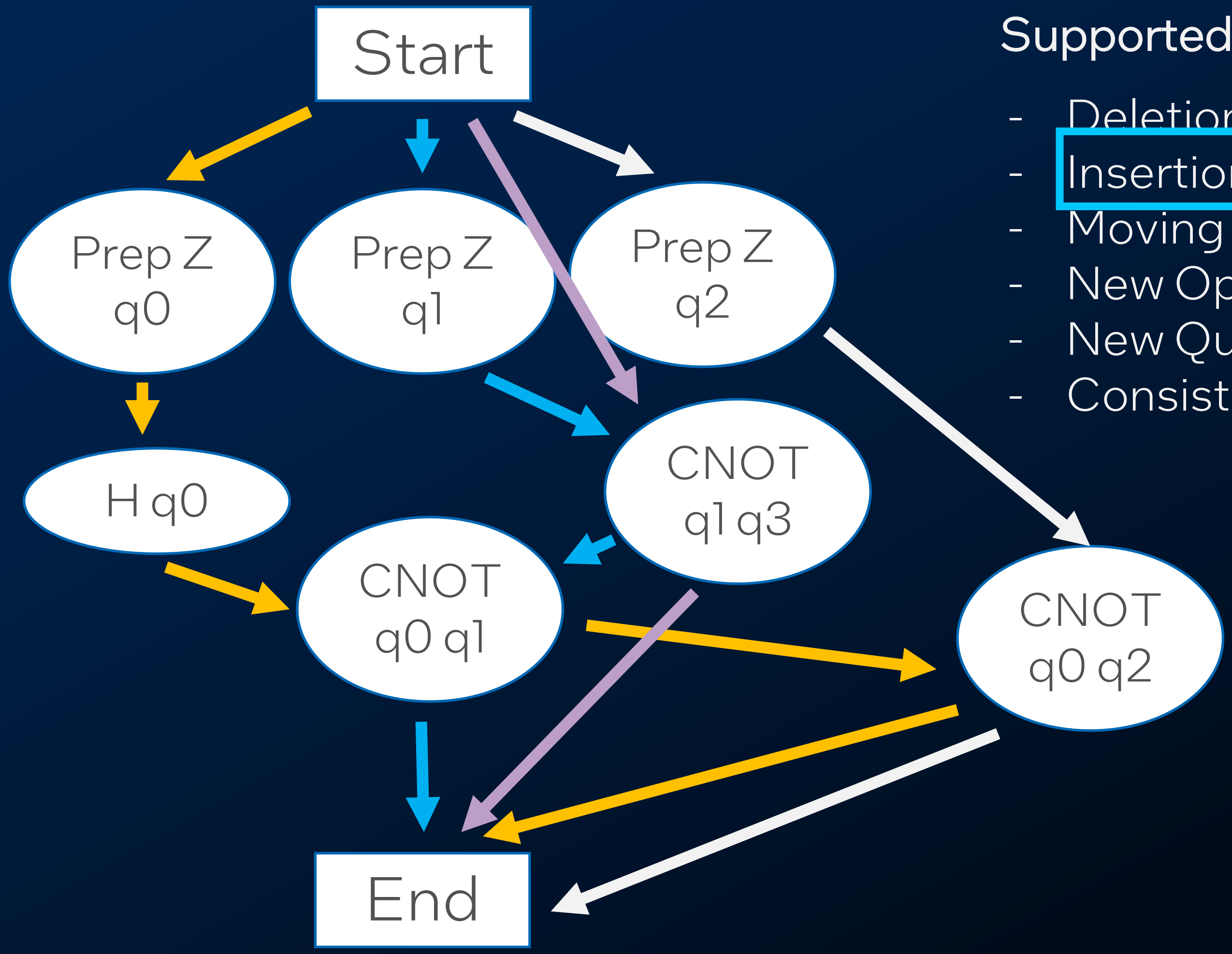
- Deletion
- Insertion
- Moving Operations
- New Operations
- New Qubits
- Consistent Iteration

```
NewOp = QuantumOp::Create(
  QuantumModule, “cnot”,
  q[1], q[3]
)
```

Circuit Object Manipulation is Reflected in IR

```
aqcc.quantum:
  . . .
  %arrayidx33 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 1
  . . .
% arrayidx21 = getelementptr inbounds [12
x i16], ptr @Qumem, i64 0, i64 1
  %arrayidx02 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 3
  call void @_Z4CNOTRtS_(ptr %arrayidx01,
ptr %arrayidx02)
  %arrayidx20 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 0
  call void @_Z4CNOTRtS_(ptr %arrayidx20,
ptr %arrayidx21)
  . . .
```

IR Form



Quantum Circuit Object

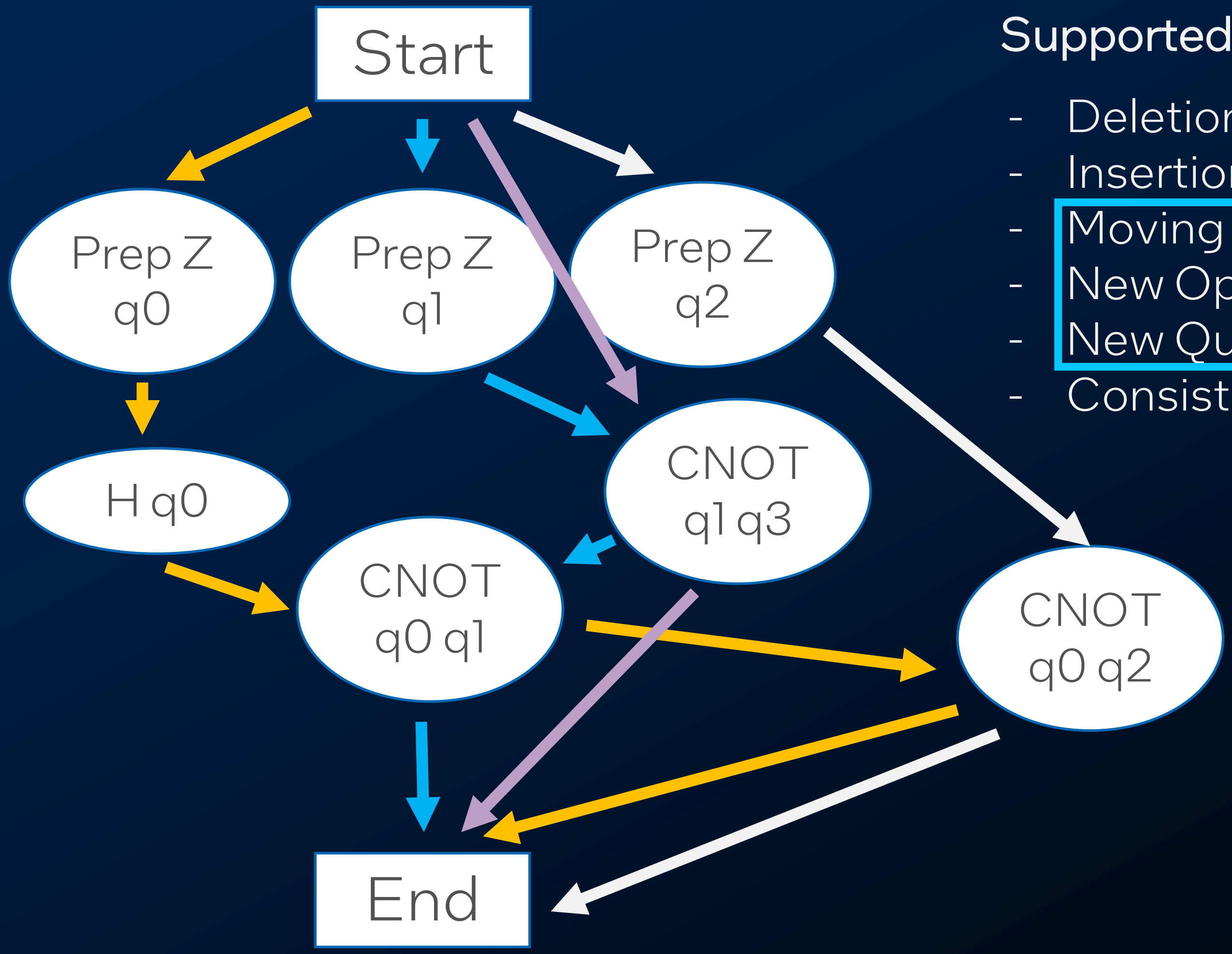
Supported Options

- Deletion
- Insertion
- Moving Operations
- New Operations
- New Qubits
- Consistent Iteration

Circuit Object Manipulation is Reflected in IR

```
aqcc.quantum:
  . . .
  %arrayidx33 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 1
  . . .
% arrayidx21 = getelementptr inbounds [12
x i16], ptr @Qumem, i64 0, i64 1
  %arrayidx02 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 3
  call void @_Z4CNOTRtS_(ptr %arrayidx01,
ptr %arrayidx02)
  %arrayidx20 = getelementptr inbounds
[12 x i16], ptr @Qumem, i64 0, i64 0
  call void @_Z4CNOTRtS_(ptr %arrayidx20,
ptr %arrayidx21)
  . . .
```

IR Form



Quantum Circuit Object

Supported Options

- Deletion
- Insertion
- Moving Operations
- New Operations
- New Qubits
- Consistent Iteration

Conclusions

intel foundry

Conclusions

- Using LLVM is a powerful part of the Intel Quantum Compiler
- Flexibility and Plugins give powerful tools to high-level developers
- LLVM is complex, the Quantum Circuit Object can alleviate some difficulties

Open Source

- Improvements in structure
- More efficient ways to use LLVM
- Quantum Optimizations from researchers

https://developer.intel.com/quantumsdk

https://github.com/intel/quantum-passes

https://github.com/intel/quantum-intrinsics

Questions?

https://developer.intel.com/quantumsdk

https://github.com/intel/quantum-passes

https://github.com/intel/quantum-intrinsics

intel foundry