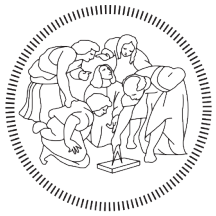

POLITECNICO
MILANO 1863

Quantum Intermediate Representation

Daniele Gazzola 10674966 Filippo Buda 10712832

Giacomo Carugati 10685017

January 2025

1 Introduction

In the world of quantum computing, one of the most used paradigm for actual computation on qubits is the gate model: logical gates are applied on qubits state in succession and with the constraint that their action must be reversible. At the end of the circuit the output qubit states are measured, in order to know the final result of the computation.

With this work, we aim to create a bridge between classical and quantum circuits through the creation of an intermediate representation that is elaborated from classical hardware descriptor languages, but is implementable on quantum hardware.

2 State of the Art

In recent years, a great deal of work has been done on the quantum programming landscape. The focus of these projects varies a lot; in the regard of compilation tool many have opted for the reuse of the existing infrastructure by embedding the domain-specific language in a widely-used programming language such as

Python, while others have instead focused their efforts in the construction of a more advanced IR quantum program, thanks to tools like MLIR.

To the best of our knowledge, all the existing work on quantum intermediate representations considers as starting point quantum high level programming languages such as Q#, OpenQasm and Qiskit. Here, we take another approach by creating an intermediate representation from a hardware descriptor language.

While this approach allows reuse of already developed techniques, it poses new challenges related to the conversion of an irreversible program paradigm to a reversible one and to the coherence in the evolution of the standard units of the computation in the two models of computation.

3 From Classical to Quantum

The main idea behind the translation is to keep the logical flow of operations unchanged. We will only convert each piece of the classical circuit into the logical correspondent in the quantum world that can be a single or a group of quantum gates.

It is important to note the fundamental difference between classical and quantum gates; the latter are reversible, while the former are irreversible. For example, the AND gate takes two input arguments and outputs one signal. The inputs cannot be reconstructed unambiguously from the output. In a quantum gate, this is possible because the number of inputs is equal to the number of inputs and can be obtained by applying its inverse gate again. The only reversible classical gate is the NOT gate, which is exactly the same in the quantum world.

As Bennett suggests [1], it is always possible to logically reverse an irreversible machine by storing each intermediate result. This is the approach that we follow in the conversion by storing each intermediate result of the quantum equivalent of a single irreversible gate in a new qubit.

This implies that many gates after the conversion will need ancillary qubits on which to write their result. Therefore, the number of qubits will always be greater than the number of original bits in the circuit. In this regard, we employ different techniques to reduce the number of ancilla qubits both during the initial construction of the MLIR from the classical circuit description and in the transformations applied cyclically at the end of the pipeline.

4 Program Pipeline

Our program is composed of a variety of programming languages, tools and libraries. The input SystemVerilog file is processed with Slang, a software library that allows us to parse it and create the corresponding Abstract Syntax Tree (AST). The syntax tree is converted from a JSON format to Dataclass objects in Python; then is traversed and each node is translated into MLIR operations thanks to xDSL, a Python library for building intermediate representations.

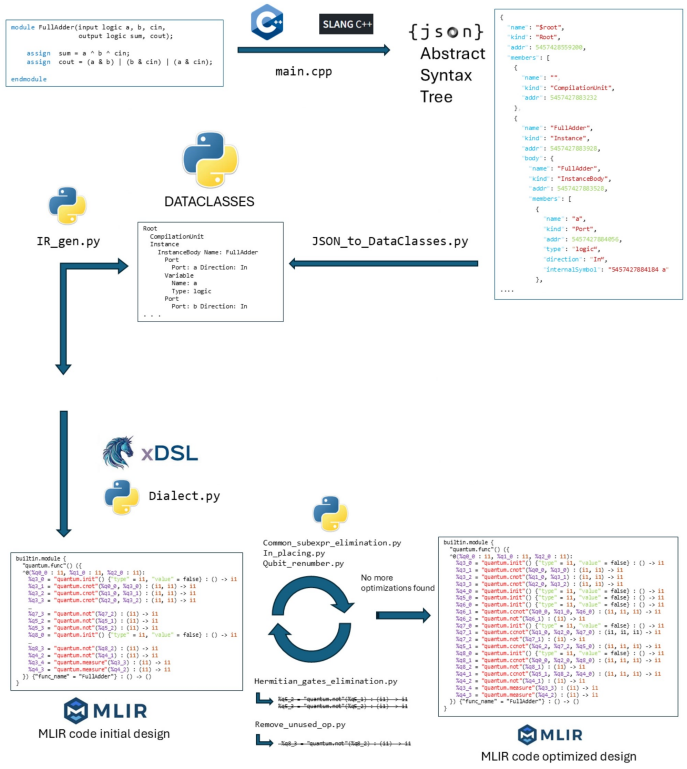

Figure 1: Tool pipeline

Once the first output is obtained, different optimization passes are applied to save on qubits and gates.

4.1 SystemVerilog Input and Slang

The first step of our pipeline is parse the SystemVerilog file, in order to create the corresponding Abstract Syntax Tree. For this purpose, we employ the `C++` library Slang [6]. Thanks to its `Driver` class we generate the AST of the input file and dump it to a JSON file.

For SLANG to correctly work, it is enough for the input file to respect the syntax rules of SystemVerilog. However, our program only supports combinatorial logic. This is because quantum circuits do not depend on a clock, therefore a sequential circuit would have no corresponding translation.

Another limitation in the input is that the entire circuit description must be contained in a single `module` construct, because it was easier for us to work with and had no semantic implications.

At last, supported gates consists only of the fundamental ones: NOT, AND,

XOR, OR as well as direct assignment of binary values to the qubits.

```
module FullAdder(input logic a, b, cin,
                 output logic sum, cout);

    assign  sum = a ^ b ^ cin;
    assign  cout = (a & b) | (b & cin) | (a & cin);

endmodule
```

```
module exampleModule(input logic a, b,c,d,
                     output logic out);
    logic temp1;

    always_comb begin
        temp1 = a & c | a;
        out = ~temp1 & (b ^ c ^ d);
    end

endmodule
```

Figure 2: Examples of supported code structure. `assign` and `always_comb` are the two ways SystemVerilog implements combinatorial statements.

4.2 From JSON to Dataclasses

Once the JSON containing the AST has been written to a file, it is subjected to further transformation. JSON is a structured data format that represents objects as key-value pairs, very much similar to the concept of Dataclass in Python. Dataclasses are special classes introduced in Python 3.7, specialized in containing data as attributes.

```
@dataclass
class BinaryOp:
    kind: str
    type: str
    op: str
    left: Union['NamedValue', 'BinaryOp']
    right: Union['NamedValue', 'BinaryOp']
    name: Optional[str] = None
    addr: Optional[int] = None
```

Figure 3: Example of a Dataclass corresponding to the BinaryOp node in the AST. Each field has a type. Some attributes can be entirely optional or assume only certain values.

Instead of working directly on the JSON tree, we developed a dedicated python program `JSON_to_Dataclass.py` that reads the JSON and translates it into nested Dataclass objects.

In this way, we build a Dataclass tree that corresponds to the original JSON one, but where each node can be accessed through an easier attribute syntax.

4.3 MLIR Generation

At this point in the program the Dataclass tree is traversed and each node is transformed in the corresponding operation defined in our dialect.

Since MLIR is not natively supported in Python, we use the xDSL library [7]; it offers classes and methods to write an intermediate representation following the MLIR paradigm. Specifically, we need to create the dialect containing our operations and the transformation passes to apply at the end of the generation.

For the design of the dialect, we adapted ideas from [4].

We decided to model each quantum gate as an operation acting on SSA values which represent qubits. Each operation will get as input at least one value and output always one qubit as a result. The result will represent the qubit that has been overwritten. The old value is then no longer used. The other inputs remain unchanged.

In general, we instantiate as many qubits (SSA values) as necessary, treating each result as a new qubit. Every time a quantum gate is created, a new qubit is instantiated for storing the result.

4.3.1 Dialect

The dialect consists of multiple operations:

- Structure Operations: `ModuleOp` and `FuncOp` are two operations used only to support code structure. They translate into the SystemVerilog concepts of `module` and `function`, but, as discussed in Section 4.1, they are used only once.

- Gate Operations: `NotOp`, `CNotOp`, `CCNotOp`, `MeasureOp`. These operations are used for the conversion from the classical gate to the quantum gate and operate directly on qubits.

  `NotOp` is the only operation acting "in-place", meaning it directly modifies the qubit is working on; others store their result in a newly initialized qubit.

  `MeasureOp` is not used in any conversion but is created at the end of the circuit to measure the output qubit and obtain the classical results.

- Service Operations: `TGateOp`, `TDaggerGateOp`, `HadamardOp`, `InitOp`. These operations do not correspond directly to the classical gates.

  `InitOp` is used to create new qubits, initializing them always to 0, while the others, although representing real quantum gates, are used only in a particular transformation of the `CCNotOp` operation for metrics evaluation purposes.

Each of the Gate and Service Operations, except for `InitOp`, takes as input one or more SSA values and returns only one SSA value corresponding to the only qubit that is written. Thus, we say that there exists a target qubit and one or more control qubits.

4.3.2 SSA Values and Qubits

Each SSA value in the intermediate representation corresponds to the state of a single qubit. The way in which we named each one of them during the MLIR generation reflects this interpretation. This idea comes from [2], which shows how it is possible to expose the data flow of the program and enhance optimizations. In our program, this way of naming the SSA values helps with their coherence assignments during the execution. In fact, the names follow the pattern %qX_Y where X is the number of the qubit itself and Y is the status number of that qubit. Therefore, Y is used to keep track of the operations that act on that qubit, where by "act" it is meant write, since read qubits do not see their status number updated. The status number is important due to

```
builtin.module {
"quantum.func"() ({
^0(%q0_0 : i1, %q1_0 : i1, %q2_0 : i1):
%q3_0 = "quantum.init"() {"type" = i1, "value" = false} : () ->
    i1
%q3_1 = "quantum.ccnot"(%q0_0, %q1_0, %q3_0) : (i1, i1, i1) -> i1
%q4_0 = "quantum.init"() {"type" = i1, "value" = false} : () ->
    i1
%q4_1 = "quantum.ccnot"(%q0_0, %q1_0, %q4_0) : (i1, i1, i1) -> i1
%q4_2 = "quantum.cnot"(%q2_0, %q4_1) : (i1, i1) -> i1
%q3_2 = "quantum.measure"(%q3_1) : (i1) -> i1
%q4_3 = "quantum.measure"(%q4_2) : (i1) -> i1
}) {"func_name" = "CseTransformation"} : () -> ()
}
```

Figure 4: Each operation returns an SSAValue that has status number equal to the target plus one. The qubit that are not written keep their original status number

the different nature of qubits and bits: once the value of a bit is modified, its previous result can be retrieved through simple rewiring, but this is not true for a qubit. Indeed, a discrepancy is present between what one can write in the input file and what can actually be done in the quantum hardware. This rule becomes important when NOT gates are introduced in the circuit.

As written in the previous subsection, the `NotOp` operation acts "in-place", meaning that the qubit is overwritten with its negated value. Therefore, a strategy must be devised in order to retrieve the original value in case it is requested again by following operations.

`NotOp` can act on two categories of targets: input variables and newly initialized qubits (that corresponds to local variables or intermediate calculation results). Two approaches are followed:

- The input variable number is known at the start of the conversion and can be used to recognize which qubits correspond to which variable.

  Since their status number always starts from zero and is only modified by `NotOp` operations (since other operations write only on newly initialized qubits), we know that if the status number is odd the current value is negated, otherwise it is the original one.

  Therefore, we can use this information to decide whether or not to add a `NotOp` in the case the original value is needed in the calculation.

- The local variables and intermediate results can be written by any operation that represents a gate. Therefore, we cannot apply the same reasoning as above. Each time a negated value is needed we negate it as usual and then once it has been used it is negated again. In this way, the original value is always available.

Examples are shown in Figure 5 and 6.

```
module notExample(input logic a, b,
                  output logic out1, out2);
    logic temp1;

    assign out1 = ~a;
    assign temp1 = a ^ b;
    assign out2 = a & ~temp1;

endmodule
```

Figure 5: Example of SystemVerilog code containing the NOT operator.

4.3.3 Gate Conversion

Each time a classical gate is encountered in the tree traversal it is transformed into its quantum equivalent. The conversions are summarized in Figure 7.

AST nodes are converted one by one as they appear in the tree. In the case of a binary operation, the left operand is visited first.

One thing to notice is the conversion we apply for the XOR gate. As can be seen in Figure 7, the quantum equivalent needs an additional qubit on which to write the results. This means that in the case of many XOR one after the other (e.g. a XOR b XOR c), as many qubits as the number of XOR would be allocated. This effect can be avoided by noticing that CNOT(X,0) is always X, so the third CNOT in the left image in 8 can be eliminated. This way, regardless of the number of concatenated XORs, we always instantiate only one new qubit.

```
builtin.module {
  "quantum.func"() ({
  ^0(%q0_0 : i1, %q1_0 : i1):
    %q2_0 = "quantum.init"() {"type" = i1, "value" = false} : ()->i1
    %q0_1 = "quantum.not"(%q0_0) : (i1) -> i1
    %q2_1 = "quantum.cnot"(%q0_1, %q2_0) : (i1, i1) -> i1
    %q0_2 = "quantum.not"(%q0_1) : (i1) -> i1
    %q3_0 = "quantum.init"() {"type" = i1, "value" = false} : ()->i1
    %q3_1 = "quantum.cnot"(%q0_2, %q3_0) : (i1, i1) -> i1
    %q3_2 = "quantum.cnot"(%q1_0, %q3_1) : (i1, i1) -> i1
    %q4_0 = "quantum.init"() {"type" = i1, "value" = false} : ()->i1
    %q0_3 = "quantum.not"(%q0_2) : (i1) -> i1
    %q3_3 = "quantum.not"(%q3_2) : (i1) -> i1
    %q3_4 = "quantum.not"(%q3_3) : (i1) -> i1
    %q4_1 = "quantum.ccnot"(%q0_3, %q3_4, %q4_0) : (i1, i1, i1) -> i1
    %q0_4 = "quantum.not"(%q0_3) : (i1) -> i1
    %q3_5 = "quantum.not"(%q3_4) : (i1) -> i1
    %q4_2 = "quantum.not"(%q4_1) : (i1) -> i1
    %q3_6 = "quantum.not"(%q3_5) : (i1) -> i1
    %q2_2 = "quantum.measure"(%q2_1) : (i1) -> i1
    %q4_3 = "quantum.measure"(%q4_2) : (i1) -> i1
  }) {"func_name" = "not"} : () -> ()
}
```

Figure 6: MLIR code employing the techniques shown above to manage NOT.

4.4 Transformations

After the whole MLIR has been generated, the final code is passed trough various transformations with the objective of reducing the number of qubit and gates, to restructure it in order to expose particular gate for metrics evaluation and to maintain it coherent.

After the completion of the MLIR generation, the program enters in a loop that continues to apply each optimization until no saving in operation and/or qubit is obtained.

Our program present a total of six transformations, further described in the following subsections: remove unused operations, common subexpression elimination, hermitian gates elimination, qubit renumber, in placing, CCNot decomposition.

4.4.1 Remove Unused Operations

This transformation simply eliminates operations that have as a result a qubit that is not used by any other operation. It is implied that the transformation acts recursively, meaning that it eliminates every operation that results "unused" after the first deletion too, and so on.

Aside from trivial cases where the input file contains unused variables, this pass becomes relevant as a cleaning procedure performed after the other transformations. In fact, when operations are eliminated, they often leave behind

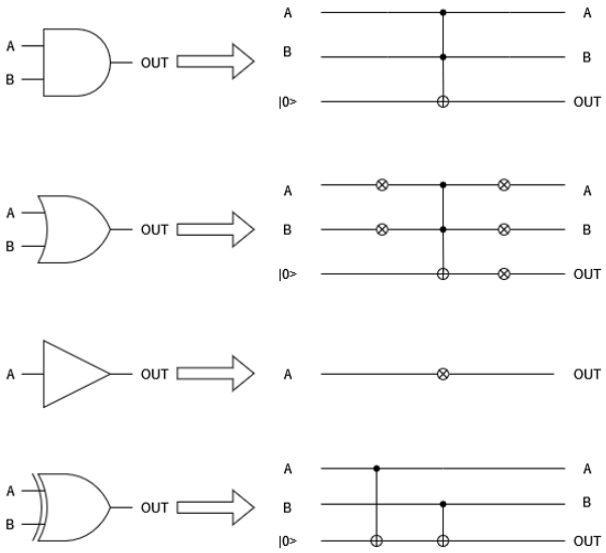

Figure 7: How each gate is transformed in its quantum equivalent.

Figure 8: a XOR b XOR c before and after the transformation

other operations whose results were used by the erased ones. These further eliminations are performed by this pass.

4.4.2 Common Subexpression Elimination

Common subexpression in general refers to the situation where the same computation is repeated in different part of the code with the same variables with the same values. The idea of the transformation is to reuse the result of the first computation in the second in order to save resources. From an MLIR point of view, this means checking if two operations generate the same result, meaning the same chain of computation is executed on two qubits. In fact, the whole history of the operands is considered in order to recognize operations doing the same computation, even if the symbolic names of the variables are different. In Figure 10 it's clear that `%q3_1` and `%q5_1` have the same value since the qubits

```
module CSE(input logic a,b,c,
           output logic out1,out2)
        assign out1 = a & b;
        assign out2 = a & b & c;
endmodule
```

Figure 9: Example of common subexpression in SystemVerilog: `out1` can be placed in the second computation instead of repeating `a & b`.

they are writing on are always initialized to 0. For this reason, as Figure 11 shows, the operation that generates `%q5_1` is eliminated, and all its uses are replaced with `%q3_1`.

```
builtin.module {
 "quantum.func"() ({
 ^0(%q0_0 : i1, %q1_0 : i1, %q2_0 : i1):
   %q3_0 = "quantum.init"() {"type" = i1, "value" = false} : ()
        -> i1
   %q3_1 = "quantum.ccnot"(%q0_0, %q1_0, %q3_0) : (i1, i1, i1) ->
        i1
   %q4_0 = "quantum.init"() {"type" = i1, "value" = false} : ()
        -> i1
   %q5_0 = "quantum.init"() {"type" = i1, "value" = false} : ()
        -> i1
   %q5_1 = "quantum.ccnot"(%q0_0, %q1_0, %q5_0) : (i1, i1, i1) ->
        i1
   %q4_1 = "quantum.ccnot"(%q5_1, %q2_0, %q4_0) : (i1, i1, i1) ->
        i1
   %q3_2 = "quantum.measure"(%q3_1) : (i1) -> i1
   %q4_2 = "quantum.measure"(%q4_1) : (i1) -> i1
 }) {"func_name" = "CseTransformation"} : () -> ()
}
```

Figure 10: Before CSE transformation

The fact that we are replacing qubits with other qubits may cause undesired effects. For example, it may happen that after this transformation a `CNotOp` ends up with the same qubit as target and control or a `CCNotOp` with the same qubit on both control entries. These situations are not transferable on real quantum hardware. In this case, we simply replace the `CNotOp` with an `InitOp` (remembering that it initializes the qubit value to zero) and the `CCNotOp` with a `CNotOp`, using only one of the two controls.

4.4.3 Hermitian Gates Elimination

All quantum gates are unitary. This means that, given a matrix U representing a quantum gate and its adjoint U^*, $U^*U = UU^* = I$ holds. In addition, CCNOT, CNOT, NOT are also Hermitian, meaning that $U = U^*$.

```
builtin.module {
  "quantum.func"() ({
  ^0(%q0_0 : i1, %q1_0 : i1, %q2_0 : i1):
    %q3_0 = "quantum.init"() {"type" = i1, "value" = false} : () -> i1
    %q3_1 = "quantum.ccnot"(%q0_0, %q1_0, %q3_0) : (i1, i1, i1) -> i1
    %q4_0 = "quantum.init"() {"type" = i1, "value" = false} : () -> i1
    %q5_0 = "quantum.init"() {"type" = i1, "value" = false} : () -> i1
    %q4_1 = "quantum.ccnot"(%q3_1, %q2_0, %q4_0) : (i1, i1, i1) -> i1
    %q3_2 = "quantum.measure"(%q3_1) : (i1) -> i1
    %q4_2 = "quantum.measure"(%q4_1) : (i1) -> i1
  }) {"func_name" = "CseTransformation"} : () -> ()
}
```

Figure 11: After CSE transformation

It follows that applying one of these gates two times on the same qubit in successive order does not modify its value.

This transformation pass checks exactly this property. The pass is trivial in the case of NOT gates (quantum and classical since their logic is the same), but it can be applied also on CNOT and CCNOT gates. Example in Figure 12.

Once this pattern is found, the two Hermitian operations are deleted.

4.4.4 XOR In-placing

It is possible to further develop the idea behind the XOR gate conversion described in Section 4.3.3. It is in fact possible to avoid initializing any new qubit at all if and only if one of the operands of the XOR chain is unused in the rest of the MLIR code and can therefore be overwritten.

This transformation pass first detects the XOR chain pattern in the MLIR, made evident by a chain of CNOT, the first writing on a newly initialized qubit and the following ones writing on the result of the previous one. Then, it checks if there is one qubit that is not used anymore (in case of more than one, only the first is considered) and modifies the chain in order for the CNOT to write on the unused qubit.

It is important to highlight that modifying one of the operands of the XOR gate in the SystemVerilog file does not contradict the statement in Subsection 4.3.2 that SSA values of input variables are never modified, except in the case of NOT gates. This holds because the transformation passes occur after the initial MLIR has been generated, and it is during this stage that the program handles the manipulation of negated values. Example shown in Figure 13.

```
module HGE(input logic a,b,c,
           output logic out);

logic temp1,temp2;

assign temp1 = a ^ a;
assign out = a & b & temp1;

endmodule
```

```
builtin.module {
  "quantum.func"() ({
  ^0(%q0_0 : i1, %q1_0 : i1, %q2_0 : i1):
    %q3_0 = "quantum.init"() {"type" = i1, "value" = false} : () -> i1
    %q3_1 = "quantum.cnot"(%q0_0, %q3_0) : (i1, i1) -> i1
    %q3_2 = "quantum.cnot"(%q0_0, %q3_1) : (i1, i1) -> i1
    %q4_0 = "quantum.init"() {"type" = i1, "value" = false} : () -> i1
    %q5_0 = "quantum.init"() {"type" = i1, "value" = false} : () -> i1
    %q5_1 = "quantum.ccnot"(%q0_0, %q1_0, %q5_0) : (i1, i1, i1) -> i1
    %q4_1 = "quantum.ccnot"(%q5_1, %q3_2, %q4_0) : (i1, i1, i1) -> i1
    %q4_2 = "quantum.measure"(%q4_1) : (i1) -> i1
  }) {"func_name" = "HGE"} : () -> ()
}
```

```
builtin.module {
  "quantum.func"() ({
  ^0(%q0_0 : i1, %q1_0 : i1, %q2_0 : i1):
    %q3_0 = "quantum.init"() {"type" = i1, "value" = false} : () -> i1
    %q4_0 = "quantum.init"() {"type" = i1, "value" = false} : () -> i1
    %q5_0 = "quantum.init"() {"type" = i1, "value" = false} : () -> i1
    %q5_1 = "quantum.ccnot"(%q0_0, %q1_0, %q5_0) : (i1, i1, i1) -> i1
    %q4_1 = "quantum.ccnot"(%q5_1, %q3_0, %q4_0) : (i1, i1, i1) -> i1
    %q4_2 = "quantum.measure"(%q4_1) : (i1) -> i1
  }) {"func_name" = "HGE"} : () -> ()
}
```

Figure 12: Example of HGE elimination, from SystemVerilog to MLIR.

```
module xorInPlace (input logic a,b,c,d,e,
                   output logic y,z);

    assign y = (a ^ b  ^ c ^ d);
    assign z = b & e;
endmodule
```

```
builtin.module {
  "quantum.func"() ({
  ^0(%q0_0 : i1, %q1_0 : i1, %q2_0 : i1, %q3_0 : i1, %q4_0 : i1):
    %q5_0 = "quantum.init"() {"type" = i1, "value" = false} : () ->
        i1
    %q5_1 = "quantum.cnot"(%q0_0, %q5_0) : (i1, i1) -> i1
    %q5_2 = "quantum.cnot"(%q1_0, %q5_1) : (i1, i1) -> i1
    %q5_3 = "quantum.cnot"(%q2_0, %q5_2) : (i1, i1) -> i1
    %q5_4 = "quantum.cnot"(%q3_0, %q5_3) : (i1, i1) -> i1
    %q6_0 = "quantum.init"() {"type" = i1, "value" = false} : () ->
        i1
    %q6_1 = "quantum.ccnot"(%q1_0, %q4_0, %q6_0) : (i1, i1, i1) -> i1
    %q6_2 = "quantum.measure"(%q6_1) : (i1) -> i1
    %q5_5 = "quantum.measure"(%q5_4) : (i1) -> i1
  }) {"func_name" = "xorInPlace"} : () -> ()
}
```

```
builtin.module {
  "quantum.func"() ({
  ^0(%q0_0 : i1, %q1_0 : i1, %q2_0 : i1, %q3_0 : i1, %q4_0 : i1):
    %q5_0 = "quantum.init"() {"type" = i1, "value" = false} : () ->
        i1
    %q0_1 = "quantum.cnot"(%q1_0, %q0_0) : (i1, i1) -> i1
    %q0_2 = "quantum.cnot"(%q2_0, %q0_1) : (i1, i1) -> i1
    %q0_3 = "quantum.cnot"(%q3_0, %q0_2) : (i1, i1) -> i1
    %q6_0 = "quantum.init"() {"type" = i1, "value" = false} : () ->
        i1
    %q6_1 = "quantum.ccnot"(%q1_0, %q4_0, %q6_0) : (i1, i1, i1) -> i1
    %q6_2 = "quantum.measure"(%q6_1) : (i1) -> i1
    %q0_4 = "quantum.measure"(%q0_3) : (i1) -> i1
  }) {"func_name" = "xorInPlace"} : () -> ()
}
```

Figure 13: Example of in-placing transformation. Having detected `%q1_0` is unused, now the CNOT gates write on this qubit.

4.4.5 Qubit Renumbering

This pass checks that the SSA value naming is coherent with the operations acting on them and eventually makes a correction. This means that each operation that writes on a target SSA value named qX_Y must have as a result an SSAValue qX_Y+1, and once the qubit has entered a successive state, the previous one must not be accessible anymore.

This pass is made necessary by the effects of the other optimization operations acting on the MLIR code. By eliminating an operation that has a result qubit used by successive operations, it is important to correct also all its uses, meaning replacing their operand with the substituted value. In addition, the qubit and status numbers of the result must be coherent with the new substituted value, thus, the need for a pass that checks and corrects any inconsistency.

4.4.6 CCNOT Decomposition

At last, the optimized MLIR code is traversed to decompose each CCNOT gate into a sequence of T, T conjugate, Hadamard and CNOT gates. This is done because in real physical quantum hardware CCNOT gates are not really implemented due to their cost, instead their behaviour is obtained by the sequence of gates displayed in Figure 14, taken from [5].

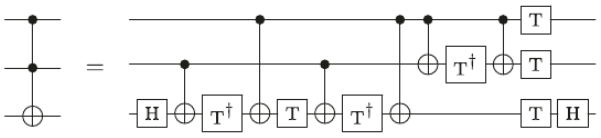

Figure 14: Conversion applied to Toffoli gates.

This way, we can measure relevant metrics such as described in the following section.

5 Testing and Metrics Evaluation

It is fundamental to test whether, given the same input, the classical and quantum circuits give the same result. We did not find any literature on theoretical methods to know if the conversion is right without performing the simulations. For this reason, we applied our algorithm on a variety of simple circuits and compared quantum and classical truth tables.

To test the output of our pipeline, we use the Python library Qiskit [3].The resulting MLIR is traversed and, based on the operation name and operands, we instantiate the correct gate in the final quantum circuit.

The quantum circuit is then simulated with all possible inputs and thus the truth table is obtained.

Instead, for the classical part, we rely on the external website edaplayground. We use the free simulator Icarus Verilog 12.0 and define a testbench to test every possible input of the circuit. At the end, we write the result on a CSV file.

Here is shown a simple example with a circuit implementing a Full Adder. The Verilog file is already present in Figure 2. It takes as input the two operands and a carry in bit, and outputs two bits, one for the sum and one for the carry out. For clarity, the truth table is shown in Figure 1.

A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 1: Full Adder truth table.

Applying our tool without the CCNOT decomposition will result in the MLIR code listed below:

```
builtin.module {
  "quantum.func"() ({
  ^0(%q0_0 : i1, %q1_0 : i1, %q2_0 : i1):
    %q3_0 = "quantum.init"() {"type" = i1, "value" = false} : () ->
        i1
    %q3_1 = "quantum.cnot"(%q0_0, %q3_0) : (i1, i1) -> i1
    %q3_2 = "quantum.cnot"(%q1_0, %q3_1) : (i1, i1) -> i1
    %q3_3 = "quantum.cnot"(%q2_0, %q3_2) : (i1, i1) -> i1
    %q4_0 = "quantum.init"() {"type" = i1, "value" = false} : () ->
        i1
    %q5_0 = "quantum.init"() {"type" = i1, "value" = false} : () ->
        i1
    %q6_0 = "quantum.init"() {"type" = i1, "value" = false} : () ->
        i1
    %q6_1 = "quantum.ccnot"(%q0_0, %q1_0, %q6_0) : (i1, i1, i1) -> i1
    %q6_2 = "quantum.not"(%q6_1) : (i1) -> i1
    %q7_0 = "quantum.init"() {"type" = i1, "value" = false} : () ->
        i1
    %q7_1 = "quantum.ccnot"(%q1_0, %q2_0, %q7_0) : (i1, i1, i1) -> i1
    %q7_2 = "quantum.not"(%q7_1) : (i1) -> i1
    %q5_1 = "quantum.ccnot"(%q6_2, %q7_2, %q5_0) : (i1, i1, i1) -> i1
    %q6_3 = "quantum.not"(%q6_2) : (i1) -> i1
    %q7_3 = "quantum.not"(%q7_2) : (i1) -> i1
    %q5_2 = "quantum.not"(%q5_1) : (i1) -> i1
    %q5_3 = "quantum.not"(%q5_2) : (i1) -> i1
    %q8_0 = "quantum.init"() {"type" = i1, "value" = false} : () ->
        i1
    %q8_1 = "quantum.ccnot"(%q0_0, %q2_0, %q8_0) : (i1, i1, i1) -> i1
    %q8_2 = "quantum.not"(%q8_1) : (i1) -> i1
    %q4_1 = "quantum.ccnot"(%q5_3, %q8_2, %q4_0) : (i1, i1, i1) -> i1
```

```
    %q5_4 = "quantum.not"(%q5_3) : (i1) -> i1
    %q8_3 = "quantum.not"(%q8_2) : (i1) -> i1
    %q4_2 = "quantum.not"(%q4_1) : (i1) -> i1
    %q3_4 = "quantum.measure"(%q3_3) : (i1) -> i1
    %q4_3 = "quantum.measure"(%q4_2) : (i1) -> i1
  }) {"func_name" = "FullAdder"} : () -> ()
}
```

In the above MLIR code, %q0_1 is A, %q1_0 is B and %q3_0 is the carry-in bit, while %q3_4 is the sum and %q4_3 is the carry-out bit. The equivalent quantum circuit is represented in the image below. In a real scenario, the

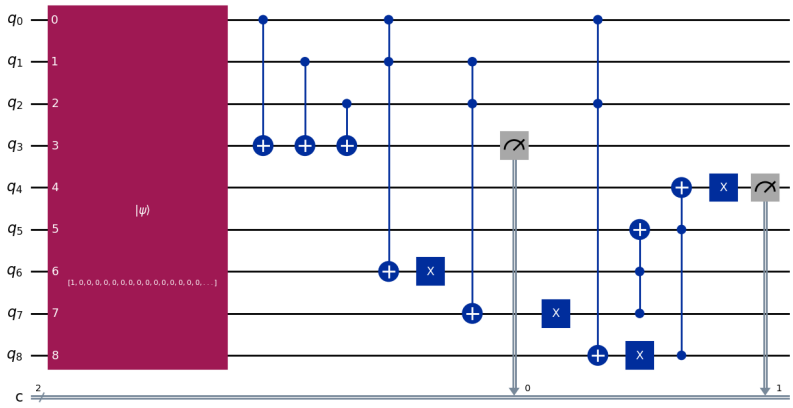

Figure 15: Resulting quantum circuit.

implemented circuit would be the one resulting from the CCNOT decomposition transformation. However, the resulting number of gates would be much larger, so, for the sake of brevity, only the initial MLIR code without transformations is shown.

5.1 Metrics

Having implemented a set of transformations and optimizations on the IR, it is interesting to evaluate their effect with a set of metrics. In particular, we take into account:

- Circuit depth: total number of serial gates.
- Circuit width: total number of instantiated qubits.
- Gate count: total number of gates.
- T-gate count: total number of T-gates.
- T-gate depth: total number of serial T-gates.

It is important to note that in order to calculate the T-gate depth metric, we used the `depth` function from Qiskit. This function, when is passed as argument a lambda function that matches only certain types of gates, returns the depth only for the specified gates. In this regard, the function will not count as part of a separate layer a gate that can be executed in parallel with another gate of the type specified in the lambda. This is the reason we do not see a great improvement in the T gate depth even if the T gate count decreases a lot.

As a first example, we will consider again the Full Adder and the MLIR code above. The comparison is made between two versions of the same MLIR: one where we apply every optimization transformation, then the CCNOT decomposition and again every optimization transformation (the "Optimized circuit"), and the other where only the CCNOT decomposition is applied (the "Original circuit").

Metric	**Original circuit**	**Optimized circuit**
Circuit Depth	45	44
Circuit Width	9	9
Gate Count	91	71
T Gate Count	35	25
T Gate Depth	18	17

Table 2: Comparison between raw circuit and the fully optimized version.

In this particular case, we notice that most of the savings derives from the reduction of the number of total gates and the number of T.gates, respectively $21,98\%$ and $28,57\%$.

5.2 Crypto Benchmarks

In this subsection we present the results of our tool in the application to some SystemVerilog files present in the GitHub directory `https://github.com/lsils/date2020_experiments` These files describe circuits much larger than the example in the previous subsection.

Here follows a table containing the measures of the metrics described above for the circuits in the aforementioned directory.

As one can see, most of the saving is obtained with the reduction in the number of qubits and in the number of gates. This can be explained by looking at the ideas behind the above transformations; most of them focus on the reduction of these quantities, while ignoring possible scheduling optimizations.

One thing we noticed is that, in some cases, it happens that the T-gate depth increases after the optimization passes. This may be due to the implicit assumption in the `depth` function we discussed earlier, together with the transformations we perform. Indeed, by eliminating operations, the schedule performed during the calculation of the depth in the basic circuit may be better in terms of parallel computation.

AES-expanded_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	1892	1812	4.22%
Circuit Width	27429	13115	52.19%
Gate Count	133506	107658	19.36%
T Gate Count	38080	30630	19.56%
T Gate Depth	609	601	1.31%

AES-non-expanded_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	3068	3341	-8.90%
Circuit Width	32308	15715	51.36%
Gate Count	163845	132412	19.18%
T Gate Count	47600	38074	20.013%
T Gate Depth	953	1079	-13.22%

DES-expanded_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	10765	9998	7.12%
Circuit Width	27285	18798	31.11%
Gate Count	269375	215555	19.98%
T Gate Count	105882	79733	24.70%
T Gate Depth	3848	3744	2.70%

DES-non-expanded_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	11980	11239	6.19%
Circuit Width	26390	18753	28.94%
Gate Count	268237	215728	19.58%
T Gate Count	105651	79719	24.54%
T Gate Depth	4344	4240	2.39%

adder_32bit_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	589	527	10.53%
Circuit Width	279	188	32.62%
Gate Count	910	659	27.58%
T Gate Count	224	128	42.86%
T Gate Depth	129	129	0%

adder_64bit_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	1127	1001	11.18
Circuit Width	541	372	31.24%
Gate Count	1786	1297	27.28%
T Gate Count	448	256	42.86%
T Gate Depth	257	256	0.38%

adder_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	2046	1832	10.46%
Circuit Width	1062	768	27.68%
Gate Count	3532	2598	26.44%
T Gate Count	896	512	42.86%
T Gate Depth	513	512	0.19%

arbiter_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	1779	1663	6.52%
Circuit Width	1566	1566	0%
Gate Count	19060	15319	19.63%
T Gate Count	8267	6148	25.63%
T Gate Depth	759	720	5.14%

bar_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	3359	2299	31.56%
Circuit Width	2823	1639	41.94%
Gate Count	17130	12611	26.38%
T Gate Count	5824	4153	28.69%
T Gate Depth	1433	1081	24.56%

cavlc_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	1356	1238	8.70%
Circuit Width	712	584	17.98%
Gate Count	8421	6797	19.29%
T Gate Count	3458	2604	24.70%
T Gate Depth	502	473	5.78%

comparator_32bit_signed_lt_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	249	233	6.43%
Circuit Width	289	191	33.91%
Gate Count	2015	1498	25.66%
T Gate Count	756	509	32.67%
T Gate Depth	92	89	3.16%

comparator_32bit_signed_lteq_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	221	213	3.62%
Circuit Width	268	194	27.61%
Gate Count	2054	1561	24.00%
T Gate Count	798	549	31.20%
T Gate Depth	85	82	3.53%

comparator_32bit_unsigned_lt_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	249	233	6.43%
Circuit Width	289	191	33.91%
Gate Count	2017	1500	25.63%
T Gate Count	756	509	32.67%
T Gate Depth	92	89	3.26%

comparator_32bit_unsigned_lteq_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	221	213	3.62%
Circuit Width	268	194	27.61%
Gate Count	2054	1561	24.00%
T Gate Count	798	549	31.20%
T Gate Depth	85	82	3.53%

ctrl_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	332	317	4.52%
Circuit Width	126	119	5.55%
Gate Count	1430	1236	13.57%
T Gate Count	595	491	17.47%
T Gate Depth	138	134	2.90%

dec_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	419	419	0%
Circuit Width	605	605	0%
Gate Count	5727	5529	3.46%
T Gate Count	2387	2259	5.36%
T Gate Depth	187	186	0.53%

div_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	43741	40616	7.14%
Circuit Width	15310	8738	42.93%
Gate Count	117772	94565	19.71%
T Gate Count	42420	32718	22.87%
T Gate Depth	14668	14361	2.09%

i2c_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	953	733	23.08%
Circuit Width	1414	1075	23.97%
Gate Count	11417	9010	21.08%
T Gate Count	4361	3189	26.87%
T Gate Depth	406	333	17.98%

int2float_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	463	423	8.64%
Circuit Width	219	151	31.05%
Gate Count	1857	1465	21.11%
T Gate Count	700	514	26.57%
T Gate Depth	152	146	3.95%

log2_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	15175	14251	6.09%
Circuit Width	28871	21391	25.91%
Gate Count	329820	270403	18.01%
T Gate Count	136052	105987	22.10%
T Gate Depth	5588	5458	2.33%

max_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	6312	5792	8.24%
Circuit Width	3052	1885	38.24%
Gate Count	19114	15540	18.70%
T Gate Count	6517	5092	21.87%
T Gate Depth	2237	2216	0.94%

md5_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	27584	24780	10.17%
Circuit Width	40346	19659	51.27%
Gate Count	222268	169204	23.87%
T Gate Count	65667	47334	27.92%
T Gate Depth	7123	7058	0.91%

mem_ctrl_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	4702	4131	12.14%
Circuit Width	11716	8834	24.60%
Gate Count	93412	73824	20.97%
T Gate Count	35791	26413	26.20%
T Gate Depth	1753	1608	8.27%

mult_32x32_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	2052	1943	5.31%
Circuit Width	6708	5207	22.38%
Gate Count	71185	59059	17.03%
T Gate Count	28749	22410	22.05%
T Gate Depth	826	803	2.78%

multiplier_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	6262	5837	6.78%
Circuit Width	20810	14472	30.46%
Gate Count	205848	167959	18.41%
T Gate Count	83580	64891	22.36%
T Gate Depth	2222	2165	2.57%

priority_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	1492	1416	5.09%
Circuit Width	621	533	14.17%
Gate Count	5679	4395	22.61%
T Gate Count	2289	1599	30.14%
T Gate Depth	520	516	0.77%

router_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	138	135	2.17%
Circuit Width	186	186	0%
Gate Count	1586	1196	24.59%
T Gate Count	672	440	34.52%
T Gate Depth	59	57	3.39%

sha-1_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	38364	35353	7.85%
Circuit Width	56803	26247	53.79%
Gate Count	292340	223253	23.63%
T Gate Count	82740	61757	25.36%
T Gate Depth	10950	10975	-0.23%

sha-256_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	37981	34517	9.12%
Circuit Width	122247	63887	47.73%
Gate Count	699062	534850	23.49%
T Gate Count	211407	151619	28.28%
T Gate Depth	11227	11042	1.65%

sin_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	4353	4263	2.07%
Circuit Width	5894	4691	20.41%
Gate Count	69440	56531	18.59%
T Gate Count	28525	21793	23.60%
T Gate Depth	1636	1639	-0.18%

sqrt_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	46834	43119	7.93%
Circuit Width	16076	9179	42.90%
Gate Count	122810	97665	20.47%
T Gate Count	43708	33122	24.22%
T Gate Depth	15285	14890	2.58%

square_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	4468	4747	-6.24%
Circuit Width	13457	8764	34.87%
Gate Count	101641	83424	17.92%
T Gate Count	36267	28219	22.19%
T Gate Depth	1641	1771	-7.92%

voter_untilsat.v			
Metric	Original	Optimized	Saving
Circuit Depth	945	869	8.04%
Circuit Width	12719	8961	29.55%
Gate Count	105932	82494	22.13%
T Gate Count	39557	27987	29.25%
T Gate Depth	330	318	3.64%

6 Conclusions

In our work we proposed new solutions and methodologies to apply already developed techniques to the case of the conversion from a classical circuit to its quantum equivalent. At the same time, we presented new ways to manage SSA values and to build a coherent flow of operations in this specific scenario. In addition, as can be seen from the tables above, we implemented a set of transformations that often manage to greatly reduce the number of qubits and operation in a circuit. We hope this work may lay the foundations for future development of tools that deal with this task.

References

[1] C. H. Bennett. "Logical Reversibility of Computation". In: *IBM Journal of Research and Development* 17.6 (1973), pp. 525–532. DOI: 10.1147/rd.176.0525.

[2] David Ittah et al. "QIRO: A Static Single Assignment-based Quantum Program Representation for Optimization". In: *ACM Transactions on Quantum Computing, Volume 3, Issue 3* (2021). URL: https://doi.org/10.1145/3491247.

[3] Ali Javadi-Abhari et al. *Quantum computing with Qiskit*. 2024. DOI: 10.48550/arXiv.2405.08810. arXiv: 2405.08810 [quant-ph].

[4] Alexander McCaskey and Thien Nguyen. *A MLIR Dialect for Quantum Assembly Languages*. 2021. arXiv: 2101.11365 [quant-ph]. URL: https://arxiv.org/abs/2101.11365.

[5] Y. Nam, N.J. Ross, and Y. et al. Su. "Automated optimization of large quantum circuits with continuous parameters." In: *npj Quantum Inf 4, 23* (2018). URL: https://doi.org/10.1038/s41534-018-0072-4.

[6] *slang - SystemVerilog Language Services*. Version 6.0. URL: https://github.com/MikePopoloski/slang.

[7] *xDSL:A Python-native SSA Compiler Framework*. Version 0.22. URL: https://github.com/xdslproject/xdsl.