Quantum Oracle Synthesis from HDL Designs via Multi Level Intermediate Representation

Giacomo Lancellotti, Filippo Buda, Giacomo Carugati, Daniele Gazzola, Alessandro Barenghi, Giovanni Agosta, and Gerardo Pelosi

Politecnico di Milano, Milano, Italy
{giacomo.lancellotti, filippo.buda, giacomo.carugati, daniele1.gazzola, alessandro.barenghi, giovanni.agosta, gerardo.pelosi}@polimi.it

***Abstract*—Quantum computing is increasingly recognized as a promising approach for tackling computationally intractable problems. However, achieving the scalability necessary for real-world applications requires substantial advancements in the quantum software stack. In this work, we introduce a compiler toolchain based on a Multi-Level Intermediate Representation (MLIR) that automatically synthesizes quantum circuits from Hardware Description Language (HDL) specifications of classical functions into quantum assembly languages. Many quantum algorithms rely on combinatorial circuits as subroutines, which traditionally require extensive resources in terms of quantum gates and qubits and are often manually optimized. Our toolchain integrates a sequence of optimization passes that combine classical compiler techniques with quantum-specific improvements, resulting in an average qubit reduction of 30% and an average gate-count reduction of 20% in widely adopted benchmark circuits, including those used in cryptographic applications.**

***Index Terms*—Quantum circuit synthesis, compiler optimizations, MLIR**

I. Introduction

Quantum computing is standing out for its potential to solve problems—such as in cryptography and simulation—more efficiently than classical methods. However, scaling quantum systems to real-world applications—potentially requiring up to 10^{12} operations [1]—faces major challenges in hardware, electronic control, error correction, and software [2]. Today quantum software stack, from SDKs to compilers, is still in its early stages. This is largely due to the lack of standardized Intermediate Representations (IRs), high-level abstractions, and stable interfaces. Popular SDKs like Qiskit [3] and Cirq [4], built on general-purpose languages such as Python, lack efficient IRs to transform and optimize quantum programs. While quantum operations often have no classical counterpart, many algorithms—such as [1], [5], [6]—use combinatorial modules like arithmetic functions and cryptographic routines. These components require many gates and qubits, significantly impacting circuit performance. Usually, many of these modules have been custom implementations derived from classical combinatorial functions and manually optimized such an approach becomes infeasible for utility-scale applications. This motivates the need for a new compiler toolchain with a custom IR and dedicated optimizations to support the automated and scalable synthesis of complex circuits.

Related Works. Notable efforts have focused on defining a standard IR for quantum computation [7], [8]. While a universally adopted quantum IR has yet to emerge, several works have explored the development of custom, domain-specific quantum compilers [9], [10]. However, none of these approaches leverage modern compiler infrastructures such as the Multi-Level Intermediate Representation (MLIR). In [11], the authors propose an MLIR-based pipeline that progressively lowers a quantum IR into an LLVM-compatible representation, thereby enabling the use of LLVM mature optimization toolchain. Building on this direction, QIRO [12] introduces a MLIR-based framework that applies dataflow code analysis and custom quantum optimization passes across two dialects to enhance quantum program execution. For the synthesis of classical combinatorial functions into quantum circuits, other works adopt dedicated logic representations—such as XOR-AND Graphs—to optimize key metrics like T-count and T-depth [13]. Additionally, compiler techniques such as graph dominator analysis have been applied to synthesize circuits with minimized qubit requirements [14]. In [15], the authors synthesized quantum oracles directly into an LLVM-based intermediate representation. While these approaches offer highly specialized optimizations, they often lack the modularity and extensibility of MLIR-based compiler toolchains.

This Work. We present a domain-specific compilation toolchain, built on the MLIR framework, that synthesizes classical functions described in a Hardware Description Language (HDL), such as SystemVerilog, into equivalent quantum circuits. MLIR models the compiler as a sequence of intermediate transformations that progressively lower a source dialect into a target dialect, where instructions can be directly mapped to those of the final quantum representation. Our pipeline leverages MLIR modularity to incorporate custom data structures and algorithms, while also reusing components from existing dialects (e.g., algebraic operations) compatible with the input. This reuse of validated infrastructure significantly improves both the efficiency and quality of the compilation process.

Contributions. We introduce a novel MLIR-based intermediate representation, designed to adapt classical combinatorial logic to quantum-specific constraints, such as reversibility. We evaluate our toolchain on a set of benchmark circuits from the literature [16] comprising cryptography-relevant and

```
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
```

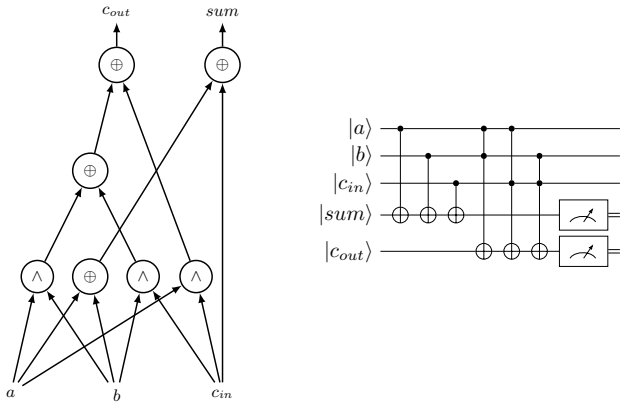

Fig. 1: From top: SystemVerilog code for a full adder, its combinational network represented as a DAG (where $\oplus$ and $\wedge$ denote XOR and AND gates, respectively), and the corresponding quantum circuit. The first three gates are CNOTs, while the last three are CCNOTs, representing the quantum equivalents of XOR and AND gates, respectively.

arithmetic kernels. We design a set of dedicated compiler optimization passes, leveraging both classical compiler optimizations such as dead code elimination and common subexpression elimination, and quantum-specific optimizations such as Hermitian gate elimination and XOR in-placing. Our approach achieves, on average, a 34.80% reduction in qubit usage, 21.48% fewer gates, and an 8.69% reduction in depth compared to unoptimized synthesized versions.

II. Background

In this section, we provide an overview of key concepts in quantum computing and the quantum circuit synthesis problem. We then introduce the MLIR framework and highlight its advantages for building domain-specific compilation toolchains.

A. Quantum Computing Basics

In quantum computing the fundamental unit of information is the *qubit*, which is defined as a two-dimensional vector in the Hilbert space $\mathcal{H} = \mathbb{C}^2$. Using bra-ket notation [17], it is a column vector given by the linear combination—i.e., the *superposition*—of two basis states $|0\rangle$ and $|1\rangle$, i.e., $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, where α and β are complex probability amplitudes that satisfy the normalization condition $|\alpha|^2 + |\beta|^2 = 1$. For a n-qubits system the state is given by $|\psi\rangle_n = \sum_{i=1}^{2^n} \alpha_i\,|i\rangle$, which corresponds to a vector in the Hilbert space of dimension $\mathbb{C}^{2^n}$. When the state of a quantum system is observed (i.e., *measured*), it collapses to one of the 2^n basis states with a probability equal to the square of the corresponding amplitude. The state of an n-qubit system is evolved by applying *quantum operators*, which are unitary matrices in $\mathbb{C}^{2^n \times 2^n}$. A matrix U is unitary if and only if $UU^\dagger = I$, where $U^\dagger$ denotes the conjugate transpose of U, which also serves as its inverse. The evolution of a quantum state is given by $|\psi_{i+1}\rangle = U\,|\psi_i\rangle$, representing the standard matrix-vector multiplication between the operator U and the state $|\psi_i\rangle$. Since quantum operators are unitary, applying them in reverse (i.e., using $U^\dagger$) allows one to recover the original state. This fundamental property implies that quantum computation is inherently *reversible*. If a sequence of k quantum operators is applied to a state, they are applied following the right-to-left order: $U_k \ldots U_1\,|\psi\rangle$.

The evolution of quantum states can be visualized using the *circuit formalism*, which resembles a classical combinatorial circuit. In this model, a qubit is represented by a horizontal wire, and quantum operators are *gates* applied to one or more qubits. The flow of computation is meant to be read from left to right. The outcome of the circuit evolution is obtained by applying a *measurement operator*, which produces a classical sequence of bits as the output. An example of a quantum circuit is reported in Fig. 1. Generally, a quantum algorithm is designed so that the evolution of the system—driven by quantum gates—increases the probability of measuring the quantum state associated with the correct solution to the problem under consideration. Many quantum algorithms rely on *oracles* to identify solution states [5], [6]. Implementing these oracles often requires quantum equivalents of classical combinatorial functions—such as arithmetic kernels [18], [19]—which can demand substantial quantum resources.

B. Quantum Circuit Synthesis

Hardware Description Languages, such as VHDL and Verilog, are used to design and simulate digital circuits. We focus on SystemVerilog—an extension of Verilog widely adopted in both academia and industry [20], [21]. This work targets combinational circuits, which are widely used as subroutines in many quantum algorithms [17]. A combinatorial function can be described by a netlist constructed from elementary logic gates such as AND, OR, and NOT. For an n-input, m-output Boolean function $f{:}\{0,1\}^n \rightarrow \{0,1\}^m$, the netlist is typically represented as a Directed Acyclic Graph (DAG), where nodes correspond to logic operations and edges represent data dependencies. An example is shown in Fig. 1. Quantum circuit synthesis consists of building a unitary operator U_f that implements the reversible version of f.

Typically, a universal gate set—such as *Clifford+T*—is used, allowing the approximation of any quantum gate to within a specified error threshold ϵ [17], much like a complete set of classical logic gates enables universal classical computation. Quantum circuit synthesis is typically modeled using the so-called *pebble game*, a mathematical abstraction on graphs to quantify resource consumption during computation. The game involves placing *pebbles* on the nodes of a DAG representing the function to be translated. Each pebble corresponds to a computational resource—specifically, a qubit—and placing a pebble on a node represents allocating a qubit to store the result computed at that node. Conversely, removing a pebble

frees the associated resource. In the reversible version of the game, introduced by Bennett [22], a node can be pebbled or unpebbled only if all its immediate predecessors are pebbled. For example, Bennett's algorithm [22] pebbles the nodes in a breadth-first, bottom-up manner. Starting from the inputs, the quantum circuit is constructed by sequentially applying the quantum gates associated with each visited node until the outputs are reached. This approach yields a circuit with minimal gate count and depth, although it does not impose constraints on the number of qubits used.

Several works have explored diverse pebbling strategies to optimize various figures of merit, such as the number of qubits used, T-count, and T-depth [13], [14], [23]. The maximum number of pebbles present on the graph at any point during computation is referred to as the *pebble number*, representing the maximum resource usage. Determining the minimum pebble number required to compute a target node in a DAG is PSPACE-complete [24], implying that no polynomial-time algorithm is known for solving it.

C. Intermediate Representations and MLIR

In modern compiler design, an Intermediate Representation (IR) serves as an abstraction layer that enables optimizations and transformations to be applied in a structured and standardized way [25]. Widely used IRs, such as LLVM IR and GCC GIMPLE, are based on the *Static Single Assignment* (SSA) form, where each variable is assigned exactly once in the program representation. This makes data dependencies explicit and simplifies static program analysis. To tackle the challenges posed by heterogeneous hardware and domain-specific languages—such as those in quantum computing, where no standard IR has yet been widely adopted [7]—the Multi-Level Intermediate Representation (MLIR) was proposed [26]. MLIR is a flexible, extensible framework designed to represent and optimize programs across multiple abstraction levels. The MLIR framework is based on the concept of *progressive lowering*, wherein high-level representations are gradually transformed into lower-level ones through a series of legal transformations with respect the target representation. The core constructs in MLIR are *operations*: each operation has a unique identifier and can take and return zero or more values. A group of operations defines a *dialect*, which has its unique namespace, types, and related semantic rules. Furthermore, the MLIR framework allows the definition and use of both source and target dialects without any predefined assumptions—such as requiring the target dialect to conform to a CPU Harvard microarchitecture, or the source to adhere to imperative, functional, or object-oriented programming paradigms.

III. Quantum MLIR stack

In this section, we present our MLIR-based compilation toolchain, featuring a custom classical-to-quantum intermediate representation, a dedicated dialect for translating classical circuits into quantum circuits, and a set of optimization passes.

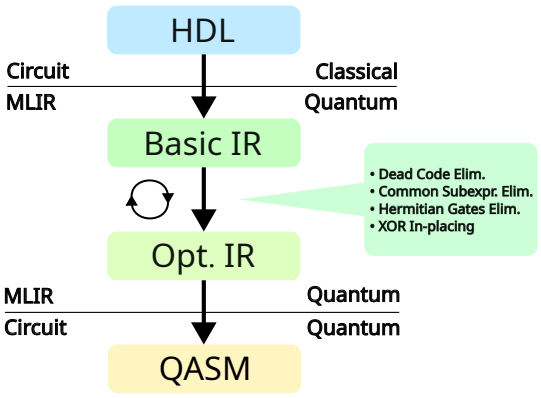

Fig. 2: Compilation flow: SystemVerilog circuit is translated into a quantum IR, optimized using classical and quantum passes, and then converted into a quantum circuit. The horizontal line indicates the switch from classical to quantum representation.

```
module test(input logic a, b, output logic out);
  logic temp;
  assign temp = a ^ b;
  assign out = a & temp;
endmodule
```

```
builtin.module{
  "quantum.func"()({
  ^0(%q0_0:i1,%q1_0:i1):
    %q2_0="quantum.init"(){"type"=i1,"value"=false}:()->
        i1
    %q2_1="quantum.cnot"(%q0_0,%q2_0):(i1, i1)->i1
    %q2_2="quantum.cnot"(%q1_0,%q2_1):(i1, i1)->i1
    %q3_0="quantum.init"(){"type"=i1,"value"=false}:()->
        i1
    %q3_1="quantum.ccnot"(%q0_0,%q2_2,%q3_0):(i1,i1,i1)->
        i1
    %q3_2="quantum.measure"(%q3_1):(i1)->i1
  }) {"func_name"="test"}:()->()
}
```

Fig. 3: Example translation of a SystemVerilog `module` into our MLIR-based quantum dialect. Input signals `a` and `b` map to qubits `%q0_0` and `%q1_0`, while auxiliary signal `temp` is `%q2_0`. The XOR is done in-place on qubit 2 using two `CNotOp` instructions, producing `%q2_2`. The AND uses a `CCNotOp`, storing its result in a newly allocated qubit `%q3_0`. Finally, `MeasureOp` reads the result, matching the output signal `out`.

A. Dialect Construction

In Fig.2, we provide a high-level overview of our toolchain. Our synthesis strategy follows Bennett's algorithm (Sec.II), applied to the Xor-And-Inverter graph of the circuit to be synthesized. Specifically, the INV gate is converted to a NOT gate, and the AND gate is mapped to a CCNOT gate, requiring two qubits for the operands and one additional qubit to store the output. For the XOR gate, we allocate two qubits for the operands and one additional qubit for the output, which

```
...
%q2_0="quantum.init"() {"type"=i1,"value"=false}:()->i1
%q2_1="quantum.cnot"(%q0_0,%q2_0):(i1,i1)->i1
%q3_0="quantum.init"(){"type"=i1,"value"=false}:()->i1
%q3_1="quantum.ccnot"(%q0_0,%q1_0,%q3_0):(i1,i1,i1)->i1
%q3_2="quantum.measure"(%q3_1):(i1)->i1
```

```
...
%q3_0="quantum.init"(){"type"=i1,"value"=false}:()->i1
%q3_1="quantum.ccnot"(%q0_0,%q1_0,%q3_0):(i1,i1,i1)->i1
%q3_2="quantum.measure"(%q3_1):(i1)->i1
```

Fig. 4: Dead Code Elimination pass. The `CNotOp` on `%q2_1` is removed as it has no effect on the program before the final `MeasureOp`. Consequently, its allocation via `InitOp` (i.e., `%q2_0`) is also eliminated.

is computed using two CNOT gates targeting the output qubit—one for each operand. Although the CNOT gate can, in principle, be performed *in place*—overwriting one of its inputs if it is not used subsequently—this information is only available after full circuit synthesis. This synthesis strategy is employed in the construction of our MLIR dialect.

For our IR we model the quantum circuit as a sequence of qubit-state updates occurring after each gate application. The IR is built using SSA form, ensuring that each *(qubit, state)* pair—denoted as `%qi_j`, where `%qi` refers to the qubit and `j` indicates the state version (starting from 0 at initialization)—is assigned exactly once throughout the program. Under this scheme, each gate application translates into a new assignment. For instance for single-qubit gates—which always act *in place* by overwriting the state of the same physical qubit—this corresponds to a state update, without the need to allocate a new qubit variable. Exposing the dataflow enables more efficient subsequent optimizations.

In our dialect, two operations capture the structure of a SystemVerilog program: `ModuleOp`, which defines a `module`—the top-level design unit with input/output ports, parameters, internal signals, and nested procedures—and `FuncOp`, which represents a `function` body restricted to combinational logic, consistent with SystemVerilog semantics. An example is shown in Fig. 3. Qubit management is handled through the `InitOp` and `MeasureOp` operations. `InitOp` allocates a new qubit and initializes it to either $|0\rangle$ or $|1\rangle$, depending on the Boolean attribute `"value"` (false for $|0\rangle$, true for $|1\rangle$). `MeasureOp` performs a measurement on a qubit, yielding a classical bit as output. Classical logic gates from the input circuit are directly translated into MLIR operations like `NotOp`, `CNotOp`, and `CCNotOp`. All gate, initialization, and measurement operations act on values of type `i1`, representing a single qubit.

B. Optimization Passes

We develop a set of optimization, legalization and analysis passes in our quantum dialect.

Dead Code Elimination. (Fig. 4.) Commonly used in classical compilers, this pass removes operations whose output qubit is never subsequently used. Since eliminating one dead

```
...
%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
```

```
...
%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"(%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
```

Fig. 5: Common Subexpression Elimination pass. The values `%q3_1` and `%q5_1` are computed through identical sequences of operations. The pass removes redundant operations producing `%q5_1`, and all of its uses (e.g., as the first operand of `%q4_1`) are replaced with the equivalent value `%q3_1`.

```
%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_1="quantum.ccnot"(%q5_1,%q3_2,%q4_0):(i1,i1,i1)->i1
%q4_2="quantum.measure"(%q4_1):(i1)->i1
```

```
%q3_0="quantum.init"(){"type"=i1, "value"=false}:()->i1
...
%q4_1="quantum.ccnot"(%q5_1,%q3_0,%q4_0):(i1,i1,i1)->i1
%q4_2="quantum.measure"(%q4_1):(i1)->i1
```

Fig. 6: Hermitian Gate Elimination example. Two consecutive `CNotOp` gates with the same control qubit `%q0` and target qubit `%q3` form an identity operation. The pass removes both gates, as they have no net effect on the quantum state.

operation may render its predecessors dead as well, the pass is applied iteratively until a fixed point is reached [27], [28]. This optimization is particularly effective after other transformations—such as operation merge or removal—as they can expose additional dead code.

Common Subexpression Elimination. (Fig. 5.) This pass identifies when two operations share an identical computation history—i.e., the same sequence of quantum gates applied to qubits with the same initialization values—and retains only one instance while eliminating redundant duplicates. However, removing these redundant operations can introduce semantic violations: for example, a cancellation may leave a `CNotOp` whose control and target qubits coincide, or a `CCNotOp` whose the two control inputs are the same qubit. Whenever such an illegal operation is detected, we replace the resulting `CNotOp` with an `InitOp`, which resets the affected qubit to the state $|0\rangle$—since, regardless of the operand values-the result of an XOR with the same values as inputs is always 0-. For a `CCNotOp`, we replace it with a `CNotOp` that uses only one of the two control qubits while preserving the original target.

Hermitian Gate Elimination. (Fig. 6.) This quantum-only

```
%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_1="quantum.ccnot"(%q1_0,%q2_0,%q4_0):(i1,i1,i1)->i1
%q3_4="quantum.measure"(%q3_3):(i1)->i1
```

```
%q0_1="quantum.cnot"(%q1_0,%q0_0):(i1,i1)->i1
%q0_2="quantum.cnot"(%q2_0,%q0_1):(i1,i1)->i1
...
%q4_1="quantum.ccnot"(%q1_0,%q2_0,%q4_0):(i1,i1,i1)->i1
%q0_3="quantum.measure"(%q0_2):(i1)->i1
```

Fig. 7: XOR In-Placing example. A sequence of `CNotOp` operations is originally applied to an intermediate qubit `%q3`. After optimization, `%q3` is eliminated, and the `CNotOp` operations with controls `%q1` and `%q2` are applied in place on `%q0`. `%q1` and `%q2` are later used as control qubits so cannot be overwritten.

```
%q2_1="quantum.ccnot"(%q0_0,%q1_0,%q2_0):(i1,i1,i1)->i1
%q1_1="quantum.tdagger"(%q1_0):(i1)->i1
```

```
%q2_1="quantum.h"(%q2_0):(i1)->i1
...
%q1_4="quantum.t"(%q1_3):(i1)->i1
%q2_9="quantum.t"(%q2_8):(i1)->i1
%q2_10="quantum.h"(%q2_9):(i1)->i1
%q1_5="quantum.tdagger"(%q1_4):(i1)->i1
```

Fig. 8: Gate decomposition example of `CCNotOp`. After the decomposition pass, the final two operations on qubit `%q1`—a `TGateOp` followed by a `TDaggerGateOp`—can be further removed by the Hermitian Gate Elimination pass.

optimization pass leverages the reversibility of quantum computation. Applying a gate twice in succession on the same qubit(s) is equivalent to the identity operation and therefore can be eliminated. We identify candidate pairs by matching both the operation type and the operand qubit indices. If the operand qubit(s) have the same index and the result of the first operation is overwritten by the second, we consider them candidates for optimization.

XOR In-Placing. (Fig. 7.) Since an XOR operation can be synthesized either out-of-place (by allocating an additional qubit to store the result) or in-place (by overwriting one of the operands), it is possible to reduce both qubit usage and gate count by rewriting the circuit in-place. No new qubit needs to be allocated if, and only if, one of the operands in an XOR sequence (i.e., an XOR chain) is no longer used later in the program and can be safely overwritten. This optimization pass first identifies XOR chains, which appear as a series of `CNotOp` operations producing output on a qubit with the same index. It then checks whether any operand in the chain is unused. If so, it rewrites the chain to use that qubit as the in-place target of all `CNotOp` operations.

Qubit-State Renaming. This serves as a legalization pass, ensuring consistency of the SSA notation throughout the entire program. In our IR, every operation that writes to a target SSA value named `%qi_j` must produce a result named `%qi_j+1`.

Gate Decomposition. (Fig. 8.) Since every quantum gate can be decomposed into a sequence of operations from a universal gate set [17], this pass acts as the main *lowering* pass towards the set of gates actually available in the target quantum computer. While not an optimization pass per se, gate decomposition can expose new opportunities for optimization. In Fig. 8, a Hermitian gate cancellation between a `TGateOp` and a `TDaggerGateOp` is identified after `CCNotOp` decomposition.

IV. Experimental Results

In this section, we summarize the tools used and outline our testing strategy. Finally, we present the performance improvements achieved by our optimization passes.

A. Experimental Setup

We first parse the SystemVerilog file using the `C++` SLANG library (version 8.0) [29] to produce the Abstract Syntax Tree (AST) of the module to be synthesized. The AST is the entry point for our dialect generation. For MLIR generation, we rely on xDSL, a Python-native compiler toolkit (version 0.22.0) [30]. The resulting quantum circuit is then generated and validated using the Qiskit SDK [31] (version 1.2.4), running on Python 3.12.3. All benchmarks [16] were executed on a Windows machine with an Intel i7-10510U processor and 16GB of RAM. The complete codebase and benchmark results are publicly available at QuantumIR.

B. Performance Analysis

We report the benchmark results using the Clifford+T gate set in Tab. I. Each optimization pass traverses the MLIR representation, applying transformations when applicable or collecting relevant information otherwise. We apply all optimization passes (except for gate decomposition) iteratively until no further improvements are observed in any metric. Then, we perform gate decomposition and run one more round of optimizations until no further improvements occur. We evaluate optimization impact using quantum circuit metrics: qubit count (width), gate count, and circuit depth, along with T-count and T-depth. We achieve an average reduction in qubits of 34.80%, an average reduction in gate count of 21.48%, and an average reduction in depth of 8.69%. Furthermore, we observe a 26.38% and a 2.77% reduction in T count and T depth respectively. These results demonstrate that qubit usage benefits the most from our optimizations, with savings exceeding 52% in the best case. This improvement is primarily due to two factors. First, several optimization passes—such as in-place XOR rewriting and dead-code elimination—are explicitly designed to reduce qubit allocation. Second, the Bennett synthesis strategy conservatively assigns a new qubit for each gate to ensure reversibility, often leading to an overestimation of the actual qubit requirements. Gate count reduction is mainly driven by Hermitian gate cancellation and common subexpression elimination, also when combined with gate decomposition. This improvement is particularly

TABLE I: Reported metrics include qubit count (width), gate count, circuit depth, T-gate count, and T-depth before and after optimization. Percentage improvements and optimization time are also shown. All circuits use the Clifford+T gate set.

	Quantum Circuit					Optimized Quantum Circuit					
Circuit	**Depth**	**Width**	**Gate**	**T-Gate**	**T-Depth**	**Depth**	**Width**	**Gate**	**T-Count**	**T-Depth**	**Time (s)**
AES-ex	1892	27429	133506	38080	609	1812 (4.23%)	13115 (52.19%)	107658 (19.36%)	30630 (19.56%)	601 (1.31%)	99.16
AES-nex	3068	32308	163845	47600	953	3341 (-8.90%)	15715 (51.36%)	132412 (19.18%)	38074 (20.01%)	1079 (-13.22%)	121.79
DES-ex	10765	27285	269375	105882	3848	9764 (9.30%)	18783 (31.16%)	213596 (20.71%)	79757 (24.67%)	3700 (3.85%)	194.39
DES-nex	11980	26390	268237	105651	4344	11094 (7.40%)	18745 (28.97%)	213904 (20.26%)	79731 (24.53%)	4215 (2.97%)	185.81
adder_64bit	1127	541	1786	448	257	1001 (11.18%)	372 (31.24%)	1297 (27.38%)	256 (42.86%)	256 (0.39%)	1.12
adder	2046	1062	3532	896	513	1832 (10.46%)	768 (27.68%)	2598 (26.44%)	512 (42.86%)	512 (0.19%)	2.27
arbiter	1779	1566	19060	8267	759	1663 (6.52%)	1566 (0.00%)	15319 (19.63%)	6148 (25.63%)	720 (5.14%)	8.83
bar	3359	2823	17130	5824	1433	2299 (31.56%)	1639 (41.94%)	12611 (26.38%)	4153 (28.69%)	1081 (24.56%)	10.42
div	43741	15310	117772	42420	14668	40517 (7.37%)	8732 (42.97%)	94473 (19.78%)	32725 (22.85%)	14339 (2.24%)	101.97
log2	15175	28871	329820	136052	5588	14055 (7.38%)	21386 (25.93%)	269555 (18.27%)	105996 (22.09%)	5438 (2.68%)	213.59
max	6312	3052	19114	6517	2237	5776 (8.49%)	1885 (38.24%)	15480 (19.01%)	5092 (21.87%)	2216 (0.94%)	12.68
md5	27584	40346	222268	65667	7123	24761 (10.23%)	19659 (51.27%)	169184 (23.88%)	47334 (27.92%)	7055 (0.95%)	172.63
mem_ctrl	4702	11716	93412	35791	1753	3939 (16.23%)	8582 (26.75%)	73154 (21.69%)	26734 (25.31%)	1576 (10.10%)	127.04
mult32x32	2052	6708	71185	28749	826	1943 (5.31%)	5207 (22.38%)	59059 (17.03%)	22410 (22.05%)	803(2.78%)	41.54
multiplier	6262	20810	205848	83580	2222	5777 (7.75%)	14468 (30.48%)	167671 (18.55%)	64898 (22.35%)	2159 (2.84%)	148.99
priority	1492	621	5679	2289	520	1416 (5.09%)	533 (14.17%)	4395 (22.61%)	1599 (30.14%)	516 (0.77%)	2.79
sha-1	38364	56803	292340	82740	10950	35310 (7.96%)	26247 (53.79%)	223187 (23.65%)	61757 (25.36%)	10972 (-0.20%)	233.74
sha-256	37981	122247	699062	211407	11227	34477 (9.23%)	63887 (47.74%)	534648 (23.52%)	151607 (28.29%)	11037 (1.69%)	584.24
sqrt	46834	16076	122810	43708	15285	42939 (8.32%)	9176 (42.92%)	97349 (20.73%)	33124 (24.22%)	14874 (2.69%)	91.43
Avg. Saving						**8.69%**	**34.80%**	**21.48%**	**26.38%**	**2.77%**	

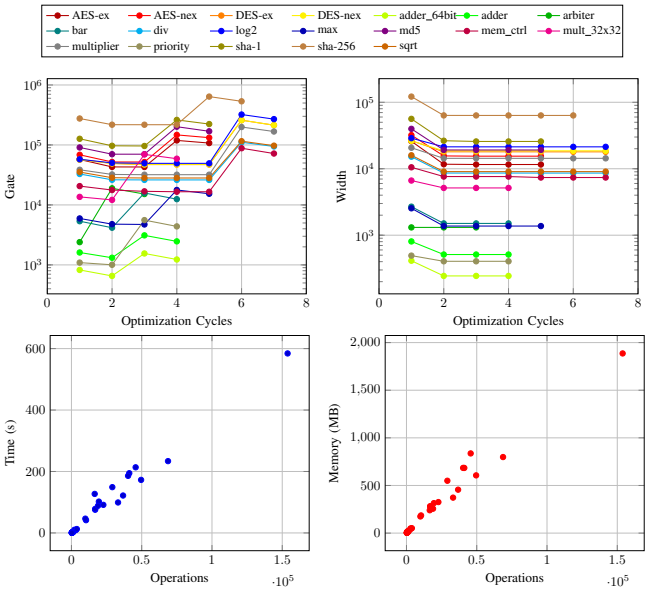

Fig. 9: Top: Gate count (left) and qubit count (right) across optimization cycles. The gate count decreases until reaching a fixed point; subsequent CCNOT decomposition causes a slight increase, followed by stabilization in the final cycle. The qubit count drops sharply after the first cycle. Bottom: Execution time (left) and memory usage (right) for the full optimization flow, both exhibiting near-linear scaling with circuit size.

marked in highly symmetric circuits such as adders. Circuit depth shows the least improvement, which is expected since our optimizations primarily target reducing qubits and gates. Introducing a gate scheduling policy could help further reduce circuit depth.

In Fig. 9, we present the evolution of gate and qubit counts across successive optimization cycles. Each benchmark requires between four and seven cycles to reach minimal resource usage. The gate count steadily decreases until the CCNOT decomposition is applied; following decomposition, one additional cycle typically suffices to reach the final count. This suggests most optimizable patterns are removed early, with post-decomposition improvements mainly due to Hermitian gate cancellations and dead-code elimination. On the right, qubit count drops sharply after the first cycle, reflecting the effectiveness of in-place rewriting and dead-code elimination early in the process. The bottom row of Fig. 9 shows execution time and memory consumption of the optimization passes, measured until convergence. Both metrics scale roughly linearly with circuit size, demonstrating the scalability of our approach.

V. Conclusions

We present an MLIR-based compiler that synthesizes SystemVerilog combinational circuits into quantum circuits compatible with existing SDKs. Our approach allows for higher-level abstractions than previous methods and supports several optimization passes, achieving up to 30% qubit reduction and 20% fewer gates compared to Bennett synthesis. Future work will extend the toolchain to support sequential logic modules.

Acknowledgements

The activity was partially funded by the *Italian Centro Nazionale di Ricerca in HPC, Big Data e Quantum computing.*

References

[1] J. Yamaguchi *et al.*, "Experiments and resource analysis of shor's factorization using a quantum simulator." Springer-Verlag, 2023, p. 119–139, [Online].

[2] M. Mohseni *et al.*, "How to build a quantum supercomputer: Scaling from hundreds to millions of qubits," 2025, [Online].

[3] A. Javadi-Abhari, M. Treinish, and Krsulich et al., "Quantum computing with Qiskit," 2024.

[4] Cirq Developers, *Cirq*. Zenodo, Apr. 2025, [Online].

[5] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *28th Annual ACM Symposium on Theory of Computing*, ser. STOC '96. ACM, 1996, p. 212–219, [Online].

[6] M. Santha, "Quantum walk based search algorithms," in *Theory and Applications of Models of Computation, 5th International Conference, TAMC 2008*, ser. LNCS, vol. 4978. Springer, 2008, pp. 31–46, [Online].

[7] F. J. Cardama, J. Vázquez-Pérez, C. Piñeiro, J. C. Pichel, T. F. Pena, and A. Gómez, "Review of intermediate representations for quantum computing," *J. Supercomput.*, vol. 81, no. 2, p. 418, 2025, [Online].

[8] A. Geller, "Introducing quantum intermediate representation (qir)," 2020, [Online].

[9] A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi, "Scaffcc: a framework for compilation and analysis of quantum computing programs," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, ser. CF '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2597917.2597939

[10] S. Sivarajah, S. Dilkes, A. Cowtan, W. Simmons, A. Edgington, and R. Duncan, "t—ket⟩: a retargetable compiler for NISQ devices," *Quantum Sci. Technol.*, vol. 6, no. 1, p. 014003, 2020.

[11] A. J. McCaskey and T. Nguyen, "A MLIR dialect for quantum assembly languages," in *IEEE International Conference on Quantum Computing and Engineering, QCE 2021*. IEEE, 2021, pp. 255–264, [Online].

[12] D. Ittah *et al.*, "Qiro: A static single assignment-based quantum program representation for optimization," *ACM Transactions on Quantum Computing*, vol. 3, no. 3, Jun. 2022, [Online].

[13] G. Meuli *et al.*, "Xor-And-Inverter Graphs for Quantum Compilation," *npj Quantum Inf.*, 2022.

[14] G. Lancellotti, G. Agosta, A. Barenghi, and G. Pelosi, "Optimizing quantum circuit synthesis with dominator analysis," in *42nd IEEE International Conference on Computer Design, ICCD 2024*. IEEE, 2024, pp. 24–27, [Online].

[15] M. Soeken and M. Mykhailova, "Automatic oracle generation in microsoft's quantum development kit using qir and llvm passes," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1363–1366. [Online]. Available: https://doi.org/10.1145/3489517.3530626

[16] E. Testa, M. Soeken, H. Riener, L. Amaru, and G. De Micheli, "A logic synthesis toolbox for reducing the multiplicative complexity in logic networks," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 568–573.

[17] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*. Cambridge university press, 2010.

[18] S. A. Cuccaro, T. G. Draper, S. Kutin, and D. P. Moulton, "A new quantum ripple-carry addition circuit," *arXiv: Quantum Physics*, 2004. [Online]. Available: https://api.semanticscholar.org/CorpusID:118445805

[19] T. Häner, S. Jaques, M. Naehrig, M. Roetteler, and M. Soeken, "Improved quantum circuits for elliptic curve discrete logarithms," in *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020, Paris, France, April 15-17, 2020, Proceedings*, ser. Lecture Notes in Computer Science, J. Ding and J. Tillich, Eds., vol. 12100. Springer, 2020, pp. 425–444. [Online]. Available: https://doi.org/10.1007/978-3-030-44223-1_23

[20] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin, *High—Level Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, 2012.

[21] S. Lahti *et al.*, "Are we there yet? a study on the state of high-level synthesis," *IEEE TCAD*, vol. 38, no. 5, pp. 898–911, 2018.

[22] C. H. Bennett, "Logical reversibility of computation," *IBM Journal of Research and Development*, vol. 17, no. 6, pp. 525–532, 1973.

[23] G. Meuli, M. Soeken, M. Roetteler, N. S. Bjørner, and G. D. Micheli, "Reversible pebbling game for quantum memory management," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, J. Teich and F. Fummi, Eds. IEEE, 2019, pp. 288–291. [Online]. Available: https://doi.org/10.23919/DATE.2019.8715092

[24] S. M. Chan, M. Lauria, J. Nordström, and M. Vinyals, "Hardness of approximation in PSPACE and separation results for pebble games," in *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, V. Guruswami, Ed. IEEE Computer Society, 2015, pp. 466–485. [Online]. Available: https://doi.org/10.1109/FOCS.2015.36

[25] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, "Compilers: Principles techniques and tools. 2007," *Google Scholar Google Scholar Digital Library Digital Library*, 2006.

[26] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 2–14.

[27] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys (CSUR)*, vol. 26, no. 4, pp. 345–420, 1994.

[28] B. G. Ryder and M. C. Paull, "Elimination algorithms for data flow analysis," *ACM Comput. Surv.*, vol. 18, no. 3, p. 277–316, Sep. 1986. [Online]. Available: https://doi.org/10.1145/27632.27649

[29] M. Popoloski, "slang - SystemVerilog Language Services," [Online].

[30] M. Fehr, M. Weber, C. Ulmann, A. Lopoukhine, M. P. Lücke, T. Degioanni, C. Vasiladiotis, M. Steuwer, and T. Grosser, "xdsl: Sidekick compilation for ssa-based compilers," in *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 179–192. [Online]. Available: https://doi.org/10.1145/3696443.3708945

[31] R. Wille, R. Van Meter, and Y. Naveh, "Ibm's qiskit tool chain: Working with and developing for real quantum computers," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 1234–1240.