Automatic oracle generation in Microsoft's Quantum Development Kit using QIR and LLVM passes

Invited paper

Mathias Soeken
Microsoft Quantum
Zürich, Switzerland
mathias.soeken@outlook.com

Mariia Mykhailova
Microsoft Quantum
Redmond, WA, USA
mamykhai@microsoft.com

ABSTRACT

Automatic oracle generation techniques can find optimized quantum circuits for classical components in quantum algorithms. However, most implementations of oracle generation techniques require that the classical component is expressed in terms of a conventional logic representation such as logic networks, truth tables, or decision diagrams. We implemented LLVM passes that can automatically generate QIR functions representing classical Q# functions into QIR code implementing such functions quantumly. We are using state-of-the-art logic optimization and oracle generation techniques based on XOR-AND graphs for this purpose. This enables not only a more natural description of the quantum algorithm on a higher level of abstraction, but also enables technology-dependent or application-specific generation of the oracles.

ACM Reference Format:
Mathias Soeken and Mariia Mykhailova. 2022. Automatic oracle generation in Microsoft's Quantum Development Kit using QIR and LLVM passes: Invited paper. In *Proceedings of Design Automation Conference (DAC '22).* ACM, New York, NY, USA, 4 pages.

1 INTRODUCTION

Implementing quantum oracles is difficult. Classical Boolean oracles are treated as black boxes in description of algorithms (see, e.g., Hamiltonian simulation [5], numerical gradient estimation [13], or amplitude amplification [8, 10]). While the *quantum parts* (reflection operator in Grover, QFT in QPE) of the algorithm are described in detail, the oracle is just a placeholder with no implementation or at best an an example for a simple classical function. In this paper, we show how to make use of the LLVM [16] infrastructure to create a QIR-based[1] tool that can automatically generate Q# operations [12, 30] for such classical oracles from Q# functions.

With the help of our approach, a developer can write a quantum program as follows:

```
namespace Operations.Classical {
  internal function Majority3(a: Bool, b: Bool, c: Bool): Bool {
    return (a or b) and (a or c) and (b or c);
  }
}

namespace Operations {
  operation Majority3(
    inputs : (Qubit, Qubit, Qubit),
    output : Qubit
  ): Unit {}

  @EntryPoint()
  operation Program(): Unit {
    use (a, b, c) = (Qubit(), Qubit(), Qubit());
    use y = Qubit();

    Majority3((a, b, c), y);
  }
}
```

The program contains an internal classical function `Majority3` that takes as input 3 Boolean arguments and returns a single Boolean value that is true if and only if the majority of the input arguments is true. The operation `Majority3`, with the same name but a different namespace, is empty and will be derived using our approach. It can then be automatically used anywhere in the code, e.g., in the `Program` operation shown in the sample.

In our approach we investigate the case in which the classical input function is defined over tuples of Boolean input arguments and returns tuples of Boolean output values. Since Q# functions are side-effect free, such functions can be represented by combinational Boolean functions $f : \mathbb{B}^n \to \mathbb{B}^m$. We will first derive such a Boolean function from the LLVM code generated for the Q# function. This function is then mapped to a quantum circuit using state-of-the-art logic synthesis based quantum compilation algorithms (see, e.g., [17–20, 22, 28]). Finally, the quantum circuit is mapped to QIR and combined with the LLVM file generated for the rest of the Q# program before linking. This enables a seamless experience, in which the user is relieved from the burden of implementing the classical function as a quantum operation. In addition, the automatic quantum compilation tools employ logic optimization to reduce the implementation cost for the oracles in terms of operation depth and qubit count.

We implemented the proposed approach in C++ and embedded it into a CMake compilation script. The complete implementation is publicly available as part of the Microsoft Quantum Development Kit samples.[2]

2 RELATED WORK

In [26], the authors have shown how to integrate phase oracle synthesis and permutation synthesis into ProjectQ [29] using RevKit [25]

[1] https://qir-alliance.org

[2] https://github.com/microsoft/Quantum/tree/main/samples/qir/oracle-generator

DAC '22, July 10–14, 2022, San Francisco, CA, USA

ACM 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530626

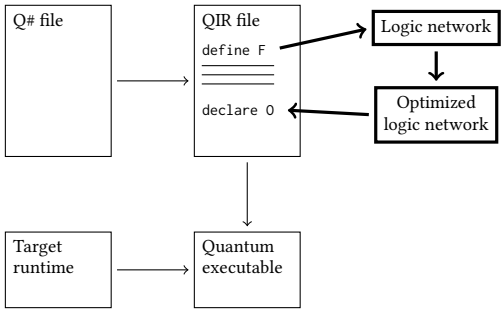

Figure 1: Proposed compilation flow

and how to integrate automatic permutation synthesis in Q# [30]. In [1], the authors describe a quantum programming flow that allows to augment openQASM programs with Verilog code, which is automatically translated into quantum circuits using tweedledum [27]. In [6], the authors present the quantum programming language silq, which maps common arithmetic operations to quantum operations and supports automatic uncomputation.

3 WORKFLOW

In this section, we illustrate the complete workflow of our proposed approach by walking through the running `Majority3` example from the introduction. Fig. 1 shows an overview of the overall flow. The Q# file is translated into QIR using the Q# compiler. This file contains an LLVM function *definition* `F` for the classical Q# function for which an oracle should be generated and an LLVM function *declaration* `0` (without code body) for the empty operation. A logic network is then created for `F`, which is further optimized using logic synthesis techniques. More precisely, we will make use of XOR-AND-inverter graphs (XAGs), which are logic networks consisting of binary XOR and binary AND gates, in which signals may be inverted. From the logic network, LLVM code is generated to be used as the code body for `0`. The resulting QIR file is compiled into a quantum executable using additional information from a target runtime. All thick arrows and boxes are contributions of our proposed flow on top of the existing QIR compilation flow. Each subsection in the remainder describes one of the arrows in the figure.

3.1 Translating LLVM into a logic network

The initial LLVM code for the `Majority3` function in Q# function looks as follows:

```
define internal i1 @Classical_Majority3(i1 %a, i1 %b, i1 %c) {
entry:
  %0 = or i1 %a, %b
  br i1 %0, label %condTrue__1, label %condContinue__1

condTrue__1:
  %1 = or i1 %a, %c
  br label %condContinue__1

condContinue__1:
  %2 = phi i1 [ %1, %condTrue__1 ], [ %0, %entry ]
  br i1 %2, label %condTrue__2, label %condContinue__2

condTrue__2:
  %3 = or i1 %b, %c
  br label %condContinue__2

condContinue__2:
  %4 = phi i1 [ %3, %condTrue__2 ], [ %2, %condContinue__1 ]
  ret i1 %4
}
```

We do not cover LLVM's syntax in detail, but describe few concepts that are important for the remainder of the paper. Every variable, function, and statement is typed. The type `i1` describes a 1-bit integer type, which can encode a Boolean value. Variable names are prefixed by a `%`. Lines such as `entry:` and `condTrue__1:` are labels and mark the beginning of a basic block. Each basic block contains a continuous sequence of statements. Statements may produce a value which is assigned to some variable. The basic block `entry:` marks the first basic block of the function. Examples for LLVM statements are `or` which computes the Boolean or of two variables and stores it into a result variable, or the `br` which jumps to some basic block unconditionally.

The translation makes use of static single assignment (SSA) ϕ-nodes [16], however, their implicitness make an automatic translation into logic networks difficult. We therefore make use of LLVM's `reg2mem` transformation pass which introduces explicit memory instructions to explicitly store intermediate variables. Since our approach only parses the transformed result to create a logic network (and not to execute it as is), these instructions will not cause any overhead in memory access in the target program. The transformed function looks as follows:

```
define internal i1 @Classical_Majority3(i1 %a, i1 %b, i1 %c) {
entry:
  %.reg2mem = alloca i1, align 1
  %.reg2mem1 = alloca i1, align 1
  %.reg2mem4 = alloca i1, align 1
  %.reg2mem6 = alloca i1, align 1
  %.reg2mem9 = alloca i1, align 1
  %.reg2mem11 = alloca i1, align 1
  %"reg2mem_alloca_point" = bitcast i32 0 to i32
  %0 = or i1 %a, %b
  store i1 %0, i1* %.reg2mem6, align 1
  %.reload8 = load i1, i1* %.reg2mem6, align 1
  br i1 %.reload8, label %condTrue__1, label %entry.
       ↪ condContinue__1_crit_edge

entry.condContinue__1_crit_edge:
  %.reload7 = load i1, i1* %.reg2mem6, align 1
  store i1 %.reload7, i1* %.reg2mem11, align 1
  br label %condContinue__1

condTrue__1:
  %1 = or i1 %a, %c
  store i1 %1, i1* %.reg2mem4, align 1
  %.reload5 = load i1, i1* %.reg2mem4, align 1
  store i1 %.reload5, i1* %.reg2mem11, align 1
  br label %condContinue__1

condContinue__1:
  %.reload12 = load i1, i1* %.reg2mem11, align 1
  store i1 %.reload12, i1* %.reg2mem1, align 1
  %.reload3 = load i1, i1* %.reg2mem1, align 1
  br i1 %.reload3, label %condTrue__2, label %condContinue__1.
       ↪ condContinue__2_crit_edge

  condContinue__1.condContinue__2_crit_edge:
  %.reload2 = load i1, i1* %.reg2mem1, align 1
  store i1 %.reload2, i1* %.reg2mem9, align 1
  br label %condContinue__2

condTrue__2:
  %2 = or i1 %b, %c
  store i1 %2, i1* %.reg2mem, align 1
```

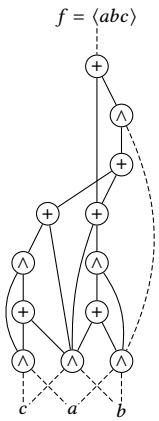

Figure 2: Initial logic network generated from LLVM code

```
  %.reload = load i1, i1* %.reg2mem, align 1
  store i1 %.reload, i1* %.reg2mem9, align 1
  br label %condContinue__2

condContinue__2:
  %.reload10 = load i1, i1* %.reg2mem9, align 1
  ret i1 %.reload10
}
```

This adds a lot of boilerplate, but most of the statements correspond to simple operations when building the logic network. Starting with nodes for the primary inputs based on the function parameters `%a`, `%b`, and `%c`, all `alloca` statements in the basic blocks will create temporary variables. These will be assigned some signal in the logic network when `load` statements are encountered. For example, in line 10 we first create a new signal `%0` by computing the OR of `%a` and `%b`, then store this value in variable `%.reg2mem6`, and load it into another variable `%.reload8`. A basic block is also assigned a signal based on the last statement in the block. The `br` statement, which is the last statement in the entry basic block, translates into a MUX operation in the logic network. Fig. 2 shows the resulting logic network after the LLVM code for the function bas been completely processed.

3.2 Optimizing the logic network

Only simple logic optimizations such as constant propagation (e.g, $1 \wedge x = x$) or structural hashing [15] (never creating nodes with the same operator and the same operands twice) are applied when creating the initial logic network. We can apply more advanced logic optimization techniques to the resulting network. It is possible to give the developer control over which cost function to assume, or even which sequence of optimization techniques to apply. However, a typical optimization flow would target reducing the number of AND gates in the logic network in favor of XOR gates, since AND gates correspond to more complicated operations both in near-term and error-corrected quantum computing devices [2]. This cost function relates to the multiplicative complexity [23] of Boolean functions. Several algorithms to reduce the number of AND gates in logic networks have been proposed [4, 7, 24, 31, 32]. In our example workflow, we employ the cut rewriting technique described in [31]. When the number of primary inputs does not exceed 8, we first collapse all outputs into their truth tables and decompose them using Shannon's decomposition rule until all internal logic nodes have 6 inputs, and use the database in [9] to map each node into its optimal XAG representation beforehand. Fig. 3 shows the logic network after optimization. Because the network has a single output and not more than 6 inputs, an optimum representation is found.

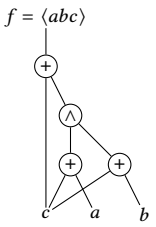

Figure 3: Optimized logic network

3.3 Compiling a logic network into QIR

There exist various techniques that describe how to map logic networks into quantum circuits, such as [17–20, 22, 28]. XAG-based methods are of particular interest due to recent optimization methods for multiplicative complexity and its close connection to resource cost of the quantum implementation [19, 20]. The mapping of an XAG into an LLVM function based on QIR is straightforward. One only needs function calls to allocate and deallocate qubits, as well as calls to CNOT and Toffoli operations. For each AND gate in the logic network, one computes the linear fanin in-place into two existing qubits and then applies a Toffoli gate controlled on these two qubits targeting a helper qubit. Eventually all outputs are copied out using CNOT gates, before all helper qubits are uncomputed by applying all operations in reverse order. For our running example the generated LLVM function looks as follows (note that we shortened some function names for the quantum operations for better readability):

```
define dso_local void @Majority3__body({ %Qubit*, %Qubit*, %Qubit*
    ↪  }* %inputs, %Qubit* %output) {
entry:
  %0 = getelementptr inbounds { %Qubit*, %Qubit*, %Qubit* }, {
      ↪ %Qubit*, %Qubit*, %Qubit* }* %inputs, i32 0, i32 0
  %a = load %Qubit*, %Qubit** %0, align 8
  %1 = getelementptr inbounds { %Qubit*, %Qubit*, %Qubit* }, {
      ↪ %Qubit*, %Qubit*, %Qubit* }* %inputs, i32 0, i32 1
  %b = load %Qubit*, %Qubit** %1, align 8
  %2 = getelementptr inbounds { %Qubit*, %Qubit*, %Qubit* }, {
      ↪ %Qubit*, %Qubit*, %Qubit* }* %inputs, i32 0, i32 2
  %c = load %Qubit*, %Qubit** %2, align 8
  %qs = call %Array* @__quantum__rt__qubit_allocate_array(i64 1)
  call void @__quantum__rt__array_update_alias_count(%Array* %qs,
      ↪ i32 1)
  call void @CNOT(%Qubit* %c, %Qubit* %a)
  call void @CNOT(%Qubit* %c, %Qubit* %b)
  %3 = call i8* @__quantum__rt__array_get_element_ptr_1d(%Array*
      ↪ %qs, i64 0)
  %4 = bitcast i8* %3 to %Qubit**
  %5 = load %Qubit*, %Qubit** %4, align 8
  call void @CCNOT(%Qubit* %a, %Qubit* %b, %Qubit* %5)
  call void @CNOT(%Qubit* %c, %Qubit* %a)
  call void @CNOT(%Qubit* %c, %Qubit* %b)
  call void @CNOT(%Qubit* %c, %Qubit* %output)
  call void @CNOT(%Qubit* %5, %Qubit* %output)
  call void @CNOT(%Qubit* %c, %Qubit* %a)
  call void @CNOT(%Qubit* %c, %Qubit* %b)
```

```
  %6 = call i8* @__quantum__rt__array_get_element_ptr_1d(%Array*
       ↪ %qs, i64 0)
  %7 = bitcast i8* %6 to %Qubit**
  %8 = load %Qubit*, %Qubit** %7, align 8
  call void @CCNOT(%Qubit* %a, %Qubit* %b, %Qubit* %8)
  call void @CNOT(%Qubit* %c, %Qubit* %a)
  call void @CNOT(%Qubit* %c, %Qubit* %b)
  call void @__quantum__rt__qubit_release_array(%Array* %qs)
  call void @__quantum__rt__array_update_alias_count(%Array* %qs,
       ↪ i32 -1)
  ret void
}
```

We can provide further control to the developer by allowing to trade off the number of helper qubits for operation count. Reversible pebble games [3, 14, 21] can be used reduce the number of qubits. This can be useful when targeting near-term devices in which the number of qubits is highly constrained. This approach can also support mapping algorithms that target the operation depth rather than operation count [11].

4 CONCLUSION

In this paper, we described a quantum compilation flow based on Q#, QIR, and LLVM, which creates quantum operations based on classical function implementations. The approach leverages logic networks and recent results in logic optimization based on multiplicative complexity. Many extensions to the proposed flow are possible, e.g., space-efficient mapping of functions defined over integer values, or automatic approximation-aware mapping of functions defined over floating-point values. Further, it is of interest to consider mapping parameterized functions, e.g., which take a dynamic sized array as input, since no straightforward logic network representation exists for such cases.

REFERENCES

[1] Matthew Amy and Vlad Gheorghiu. 2020. staq—A full-stack quantum processing toolkit. *Quantum Science and Technology* 5, 3 (jun 2020), 034016. https://doi.org/10.1088/2058-9565/ab9359

[2] Matthew Amy, Dmitri Maslov, Michele Mosca, and Martin Roetteler. 2013. A Meet-in-the-Middle Algorithm for Fast Synthesis of Depth-Optimal Quantum Circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems* 32, 6 (2013), 818–830. https://doi.org/10.1109/TCAD.2013.2244643

[3] Charles Henry Bennett. 1989. Time/Space Trade-Offs for Reversible Computation. *SIAM J. Comput.* 18, 4 (1989), 766–776. https://doi.org/10.1137/0218053

[4] Anna Bernasconi, Stelvio Cimato, Valentina Ciriani, and Maria Chiara Molteni. 2020. Multiplicative Complexity of Autosymmetric Functions: Theory and Applications to Security. In *Design Automation Conference*. IEEE, 1–6. https://doi.org/10.1109/DAC18072.2020.9218492

[5] Dominic W. Berry, Andrew M. Childs, and Robin Kothari. 2015. Hamiltonian Simulation with Nearly Optimal Dependence on all Parameters. In *Foundations of Computer Science*. IEEE Computer Society, 792–809. https://doi.org/10.1109/FOCS.2015.54

[6] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin T. Vechev. 2020. Silq: a high-level quantum language with safe uncomputation and intuitive semantics. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 286–300. https://doi.org/10.1145/3385412.3386007

[7] Joan Boyar, Philip Matthews, and René Peralta. 2013. Logic Minimization Techniques with Applications to Cryptology. *Journal of Cryptology* 26, 2 (2013), 280–312. https://doi.org/10.1007/s00145-012-9124-7

[8] Gilles Brassard, Peter Hoyer, Michele Mosca, and Alain Tapp. 2002. Quantum amplitude amplification and estimation. *Contemp. Math.* 305 (2002), 53–74.

[9] Çagdas Çalik, Meltem Sönmez Turan, and René Peralta. 2019. The multiplicative complexity of 6-variable Boolean functions. *Cryptography and Communications* 11, 1 (2019), 93–107. https://doi.org/10.1007/s12095-018-0297-2

[10] Lov K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. In *Symposium on Theory and Computing*. 212–219. https://doi.org/10.1145/237814.237866

[11] Thomas Häner and Mathias Soeken. 2022. Lowering the T-Depth of Quantum Circuits via Logic Network Optimization. *ACM Transactions on Quantum Computing* 3, 2 (2022), 15 pages. https://doi.org/10.1145/3501334

[12] Bettina Heim, Mathias Soeken, Sarah Marshall, Chris Granade, Martin Roetteler, Alan Geller, Matthias Troyer, and Krysta Svore. 2020. Quantum programming languages. *Nature Reviews Physics* (2020). https://doi.org/10.1038/s42254-020-00245-7

[13] Stephen P. Jordan. 2005. Fast Quantum Algorithm for Numerical Gradient Estimation. *Physical Review Letters* 95 (Jul 2005), 050501. Issue 5. https://doi.org/10.1103/PhysRevLett.95.050501

[14] Balagopal Komarath, Jayalal Sarma, and Saurabh Sawlani. 2015. Reversible Pebble Game on Trees. In *Int'l Conf. on Computing and Combinatorics*. 83–94. https://doi.org/10.1007/978-3-319-21398-9_7

[15] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K. Ganai. 2002. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Trans. on CAD of Integrated Circuits and Systems* 21, 12 (2002), 1377–1394. https://doi.org/10.1109/TCAD.2002.804386

[16] Chris Lattner and Vikram Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *Int'l Symp. on Code Generation and Optimization*. https://doi.org/10.1109/CGO.2004.1281665

[17] Igor L. Markov and Mehdi Saeedi. 2012. Constant-optimized quantum circuits for modular multiplication and exponentiation. *Quantum Information and Computation* 12, 5&6 (2012), 361–394.

[18] Igor L. Markov and Mehdi Saeedi. 2013. Faster quantum number factoring via circuit synthesis. *Physical Review A* 87 (2013), 012310. Issue 1. https://doi.org/10.1103/PhysRevA.87.012310

[19] Giulia Meuli, Mathias Soeken, Earl Campbell, Martin Roetteler, and Giovanni De Micheli. 2019. The Role of Multiplicative Complexity in Compiling Low T-count Oracle Circuits. In *Int'l Conf. on Computer-Aided Design*. 1–8. https://doi.org/10.1109/ICCAD45719.2019.8942093

[20] Giulia Meuli, Mathias Soeken, and Giovanni De Micheli. 2022. Xor-And-Inverter Graphs for Quantum Compilation. *npj Quantum Information* 8, 7 (2022), 431. https://doi.org/10.1038/s41534-021-00514-y

[21] Giulia Meuli, Mathias Soeken, Martin Roetteler, Nikolaj Bjørner, and Giovanni De Micheli. 2019. Reversible Pebbling Game for Quantum Memory Management. In *Design, Automation and Test in Europe*. 288–291. https://doi.org/10.23919/DATE.2019.8715092

[22] Mariusz Rawski. 2015. Application of Functional Decomposition in Synthesis of Reversible Circuits. In *Int'l Conf. on Reversible Computation*. 285–290. https://doi.org/10.1007/978-3-319-20860-2_20

[23] Claus-Peter Schnorr. 1988. The Multiplicative Complexity of Boolean Functions. In *Int'l Conf. on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*. 45–58. https://doi.org/10.1007/3-540-51083-4_47

[24] Mathias Soeken. 2020. Determining the Multiplicative Complexity of Boolean Functions using SAT. *arXiv preprint arXiv:2005.01778* (2020).

[25] Mathias Soeken, Stefan Frehse, Robert Wille, and Rolf Drechsler. 2011. RevKit: An Open Source Toolkit for the Design of Reversible Circuits. In *Int'l Workshop on Reversible Computation*. 64–76. https://doi.org/10.1007/978-3-642-29517-1_6

[26] Mathias Soeken, Thomas Häner, and Martin Roetteler. 2018. Programming quantum computers using design automation. In *Design, Automation and Test in Europe*. 137–146. https://doi.org/10.23919/DATE.2018.8341993

[27] Mathias Soeken, Heinz Riener, Winston Haaswijk, Eleonora Testa, Bruno Schmitt, Giulia Meuli, Fereshte Mozafari, and Giovanni De Micheli. 2018. The EPFL logic synthesis libraries. *arXiv preprint arXiv:1805.05121v2* (2018).

[28] Mathias Soeken, Martin Roetteler, Nathan Wiebe, and Giovanni De Micheli. 2019. LUT-Based Hierarchical Reversible Logic Synthesis. *IEEE Trans. on CAD of Integrated Circuits and Systems* 38, 9 (2019), 1675–1688. https://doi.org/10.1109/TCAD.2018.2859251

[29] Damian S. Steiger, Thomas Haener, and Matthias Troyer. 2016. ProjectQ: An Open Source Software Framework for Quantum Computing. *arXiv preprint arXiv:1612.08091* (2016).

[30] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. 2018. Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. In *Real World Domain Specific Languages Workshop*. 7:1–7:10. https://doi.org/10.1145/3183895.3183901

[31] Eleonora Testa, Mathias Soeken, Luca G. Amarù, and Giovanni De Micheli. 2019. Reducing the Multiplicative Complexity in Logic Networks for Cryptography and Security Applications. In *Design Automation Conference*. 74. https://doi.org/10.1145/3316781.3317893

[32] Eleonora Testa, Mathias Soeken, Heinz Riener, Luca Gaetano Amarù, and Giovanni De Micheli. 2020. A logic synthesis toolbox for reducing the multiplicative complexity in logic networks. In *Design, Automation and Test in Europe*.