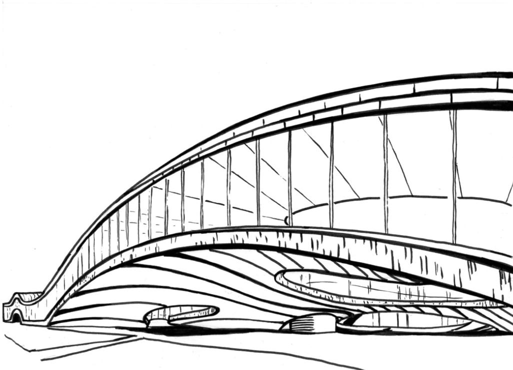
Reversible logic synthesis and RevKit

Mathias Soeken

Integrated Systems Laboratory, EPFL, Switzerland

mathias.soeken@epfl.ch msoeken.github.io msoeken/cirkit



Introduction and background

Reversible logic synthesis

Introduction into RevKit

Introduction and background

Reversible logic synthesis

Introduction into RevKit

Quantum computing is getting real

- 17-qubit quantum computer from IBM based on superconducting qubits (16-qubit version available via cloud service)
- 9-qubit quantum computer from Google based on superconducting circuits
- 5-qubit quantum computer at University of Maryland based on ion traps
- Microsoft is investigating topological quantum computers
- Intel is investigating silicon-based qubits

Quantum computing is getting real

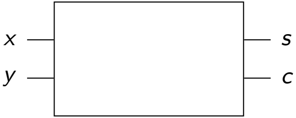
- 17-qubit quantum computer from IBM based on superconducting qubits (16-qubit version available via cloud service)
- 9-qubit quantum computer from Google based on superconducting circuits
- 5-qubit quantum computer at University of Maryland based on ion traps
- Microsoft is investigating topological quantum computers
- Intel is investigating silicon-based qubits

- "Quantum supremacy" experiment may be possible with $\approx$50 qubits (45-qubit simulation has been performed classically)
- Smallest practical problems require $\approx$100 (logical) qubits

Challenges in logic synthesis for quantum computing

1. Quantum computers process qubits not bits

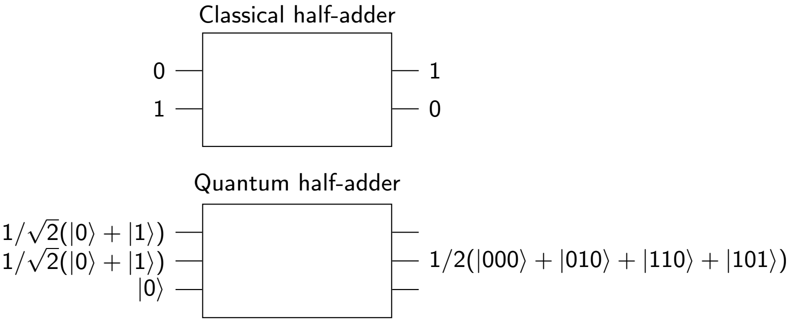
Classical half-adder



Quantum half-adder

Challenges in logic synthesis for quantum computing

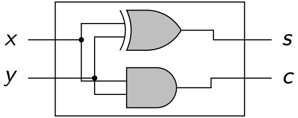
1. Quantum computers process qubits not bits



Challenges in logic synthesis for quantum computing

1. Quantum computers process qubits not bits
2. All qubit operations, called quantum gates, must be reversible

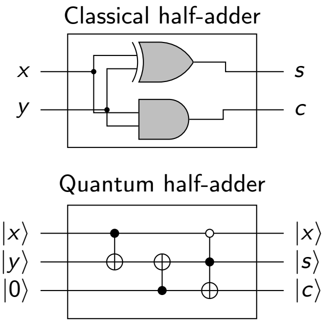
Classical half-adder



Quantum half-adder

Challenges in logic synthesis for quantum computing

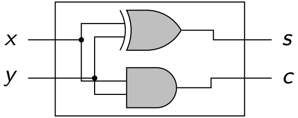
1. Quantum computers process qubits not bits
2. All qubit operations, called quantum gates, must be reversible



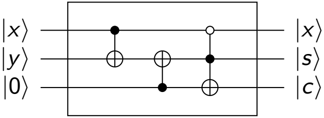
Challenges in logic synthesis for quantum computing

1. Quantum computers process qubits not bits
2. All qubit operations, called quantum gates, must be reversible
3. Standard gate library for today's physical quantum computers is non-trivial

Classical half-adder



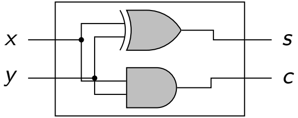
Quantum half-adder



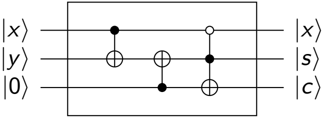
Challenges in logic synthesis for quantum computing

1. Quantum computers process qubits not bits
2. All qubit operations, called quantum gates, must be reversible
3. Standard gate library for today's physical quantum computers is non-trivial
4. Circuit is not allowed to produce intermediate results, called garbage qubits

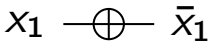
Classical half-adder



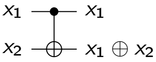
Quantum half-adder



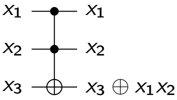
Reversible gates



NOT

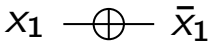


CNOT

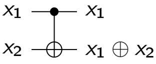


Toffoli

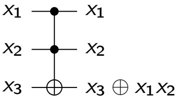
Reversible gates



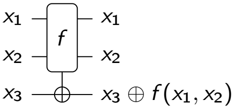
NOT



CNOT

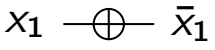


Toffoli

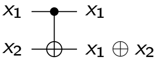


Single-target

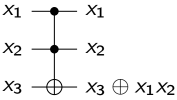
Reversible gates



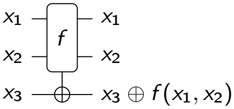
NOT



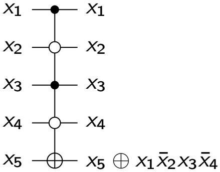
CNOT



Toffoli

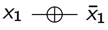


Single-target

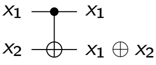


Multiple-controlled Toffoli

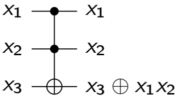
Reversible gates



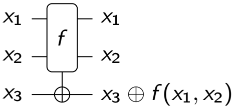
NOT



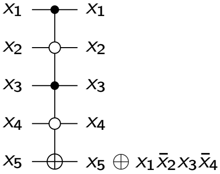
CNOT



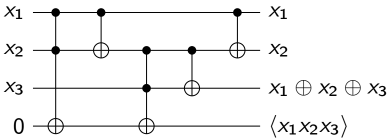
Toffoli



Single-target



Multiple-controlled Toffoli



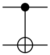
Full adder

Quantum gates

- Qubit is vector $|\varphi\rangle = \binom{\alpha}{\beta}$ with $|\alpha^2| + |\beta^2| = 1$.
- Classical 0 is $|0\rangle = \binom{1}{0}$; Classical 1 is $|1\rangle = \binom{0}{1}$

Quantum gates

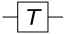
- Qubit is vector $|\varphi\rangle = \binom{\alpha}{\beta}$ with $|\alpha^2| + |\beta^2| = 1$.
- Classical 0 is $|0\rangle = \binom{1}{0}$; Classical 1 is $|1\rangle = \binom{0}{1}$

$|\varphi_1\rangle$, $|\varphi_2\rangle$  $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} |\varphi_1\varphi_2\rangle$

CNOT

$|\varphi\rangle$  $\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} |\varphi\rangle$

Hadamard

$|\varphi\rangle$  $\begin{pmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{pmatrix} |\varphi\rangle$

T

Composing quantum gates

- Applying a quantum gate to a quantum state (matrix-vector multiplication)

$$|\varphi\rangle = \left(\begin{smallmatrix}\alpha \\ \beta\end{smallmatrix}\right) \;-\!\boxed{U}\!-\; U|\varphi\rangle$$

Composing quantum gates

- Applying a quantum gate to a quantum state (matrix-vector multiplication)

$$|\varphi\rangle = \left(\begin{smallmatrix}\alpha\\ \beta\end{smallmatrix}\right) -\boxed{U}- \; U|\varphi\rangle$$

- Applying quantum gates in sequence (matrix product)

$$|\varphi\rangle = \left(\begin{smallmatrix}\alpha\\ \beta\end{smallmatrix}\right) -\boxed{U_1}-\boxed{U_2}- \; (U_2 U_1)|\varphi\rangle$$

Composing quantum gates

- Applying a quantum gate to a quantum state (matrix-vector multiplication)

$$|\varphi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \;-\!\boxed{U}\!-\; U|\varphi\rangle$$

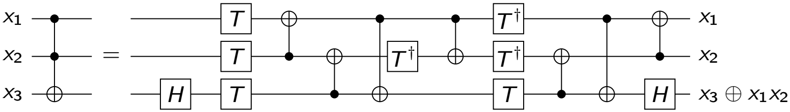
- Applying quantum gates in sequence (matrix product)

$$|\varphi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \;-\!\boxed{U_1}\!-\!\!-\!\boxed{U_2}\!-\; (U_2 U_1)|\varphi\rangle$$

- Applying quantum gates in parallel (Kronecker product)

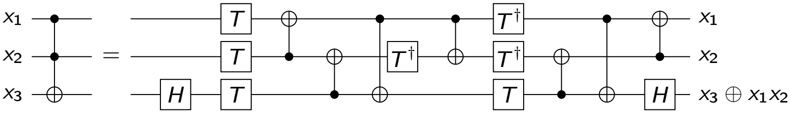
$$\left.\begin{array}{l} |\varphi_1\rangle = \begin{pmatrix} \alpha_1 \\ \beta_1 \end{pmatrix} \;-\!\boxed{U_1}\!- \\ |\varphi_2\rangle = \begin{pmatrix} \alpha_2 \\ \beta_2 \end{pmatrix} \;-\!\boxed{U_2}\!- \end{array}\right\} (U_1 \otimes U_2)|\varphi_1\varphi_2\rangle$$

Mapping Toffoli gates

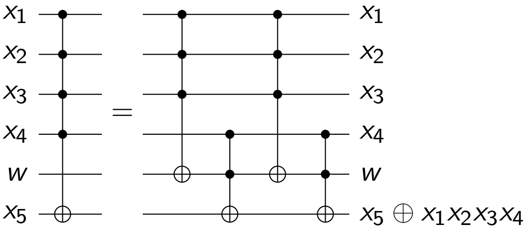


Clifford+T circuit [Amy *et al.*, *TCAD* **32**, 2013]

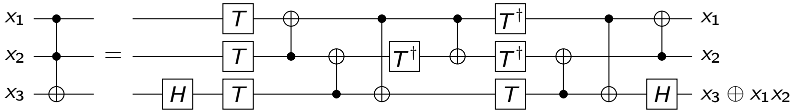
Mapping Toffoli gates



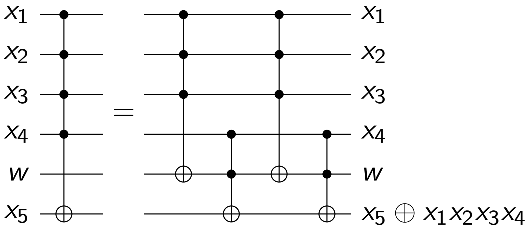
Clifford+T circuit [Amy *et al.*, *TCAD* **32**, 2013]



Mapping Toffoli gates



Clifford+T circuit [Amy *et al.*, *TCAD* **32**, 2013]



Costs are number of qubits and number of T gates

RevKit

- Open source C++ framework for reversible logic synthesis (since 2009)

RevKit

- Open source C++ framework for reversible logic synthesis (since 2009)
- Implemented as add-on in CirKit, an open source C++ framework for logic synthesis

RevKit

- Open source C++ framework for reversible logic synthesis (since 2009)
- Implemented as add-on in CirKit, an open source C++ framework for logic synthesis
- Provides command-line interface shell (CLI) and API

RevKit

- Open source C++ framework for reversible logic synthesis (since 2009)
- Implemented as add-on in CirKit, an open source C++ framework for logic synthesis
- Provides command-line interface shell (CLI) and API
- CLI allows batch processing and can be used via Python API

RevKit

- Open source C++ framework for reversible logic synthesis (since 2009)
- Implemented as add-on in CirKit, an open source C++ framework for logic synthesis
- Provides command-line interface shell (CLI) and API
- CLI allows batch processing and can be used via Python API
- No Python API for C++ API

RevKit

- Open source C++ framework for reversible logic synthesis (since 2009)
- Implemented as add-on in CirKit, an open source C++ framework for logic synthesis
- Provides command-line interface shell (CLI) and API
- CLI allows batch processing and can be used via Python API
- No Python API for C++ API
- Implement core functionality in C++ and expose it via CLI commands

RevKit [Installation]

- Obtain from Github: github.com/msoeken/cirkit

RevKit [Installation]

- Obtain from Github: github.com/msoeken/cirkit
- Works smoothly on modern Mac OS and Linux distributions

RevKit [Installation]

- Obtain from Github: github.com/msoeken/cirkit
- Works smoothly on modern Mac OS and Linux distributions
- Works on Windows OS using Ubuntu subsystem

RevKit [Installation]

- Obtain from Github: github.com/msoeken/cirkit
- Works smoothly on modern Mac OS and Linux distributions
- Works on Windows OS using Ubuntu subsystem
- Install dependencies via package manager (in Mac OS, e.g., brew)

RevKit [Installation]

- Obtain from Github: github.com/msoeken/cirkit
- Works smoothly on modern Mac OS and Linux distributions
- Works on Windows OS using Ubuntu subsystem
- Install dependencies via package manager (in Mac OS, e.g., brew)
- More details: msoeken.github.io/revkit.html

RevKit [Generalities]

- RevKit is commands plus stores
- Stores for each relevant data structure: e.g., reversible circuits, reversible truth tables, AND-inverter graphs, binary decision diagrams, . . .
- Stores can contain several instances, commands work on current element

```
revkit> read_real -s "t a b c, f b a c"
revkit> store -c
[i] circuits in store:
  *  0: 3 lines, 2 gates
revkit> tof
revkit> ps -c
Lines:        3
Gates:        4
T-count:      14
Logic qubits: 4
revkit> write_liquid file.fs
```

Introduction and background

Reversible logic synthesis

Introduction into RevKit

Reversible logic synthesis

$$\text{Boolean function} \xrightarrow[\text{embedding}]{} \text{Rev. function} \xrightarrow[\text{synthesis}]{} \text{Rev. circuit}$$

- What is the input representation?
- Is the input representation reversible or irreversible?
- Explicit vs. implicit embedding

Reversible logic synthesis

$$\text{Boolean function} \xrightarrow[\text{embedding}]{\text{optimum}} \text{Rev. function} \xrightarrow[\text{synthesis}]{} \text{Rev. circuit}$$

- What is the input representation?
- Is the input representation reversible or irreversible?
- Explicit vs. implicit embedding

Reversible logic synthesis

$$\text{Boolean function} \xrightarrow[\text{embedding}]{\text{optimum}} \text{Rev. function} \xrightarrow[\text{synthesis}]{\text{ancilla-free}} \text{Rev. circuit}$$

- What is the input representation?
- Is the input representation reversible or irreversible?
- Explicit vs. implicit embedding

Embedding of irreversible functions

c	x	y	c	s
	0	0	0	0
	0	1	0	1
	1	0	0	1
	1	1	1	0

Embedding of irreversible functions

c	x	y	c	s	x
	0	0	0	0	0
	0	1	0	1	0
	1	0	0	1	1
	1	1	1	0	1

- Add additional outputs to disambiguate duplicate output pattern, preferably inputs

Embedding of irreversible functions

c	x	y	c	s	x
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	1

- Add additional outputs to disambiguate duplicate output pattern, preferably inputs
- Add additional inputs to match number of outputs, preferably constants

Embedding of irreversible functions

c	x	y	c	s	x
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0			
1	0	1			
1	1	0			
1	1	1			

- Add additional outputs to disambiguate duplicate output pattern, preferably inputs
- Add additional inputs to match number of outputs, preferably constants
- Optional: assign output pattern to new input pattern

Embedding of irreversible functions

c	x	y	c	s	x
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0			0
1	0	1			0
1	1	0			1
1	1	1			1

- Add additional outputs to disambiguate duplicate output pattern, preferably inputs
- Add additional inputs to match number of outputs, preferably constants
- Optional: assign output pattern to new input pattern

Embedding of irreversible functions

c	x	y	c	s	x
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	1	1	0
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	1	1

- Add additional outputs to disambiguate duplicate output pattern, preferably inputs
- Add additional inputs to match number of outputs, preferably constants
- Optional: assign output pattern to new input pattern

Embedding of irreversible functions

c	x	y	c	s	x
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	1	1	0
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	1	1

- Add additional outputs to disambiguate duplicate output pattern, preferably inputs
- Add additional inputs to match number of outputs, preferably constants
- Optional: assign output pattern to new input pattern
- Upper bound: Number of original inputs + number of original outputs (all inputs are preserved)

Embedding of irreversible functions

c	x	y	c	s	x
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	1	1	0
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	1	1

- Add additional outputs to disambiguate duplicate output pattern, preferably inputs
- Add additional inputs to match number of outputs, preferably constants
- Optional: assign output pattern to new input pattern
- Upper bound: Number of original inputs + number of original outputs (all inputs are preserved)
- Computing whether optimum embedding is possible with ℓ additional lines is coNP-hard

Reversible synthesis classification

	line opt.	*gate opt.*	**nonreversible func.**	**reversible func.**
functional	✔	✔		SAT-based Enumerative
functional	✔	✘		Transformation-based Cycle-based Decomposition-based Metaheuristic Greedy
structural	✘	✘	ESOP-based Hierarchical Building block	

SAT-based exact synthesis

- **Idea:** Solve question *"does there exist a reversible circuit realizing f with r gates?"* as SAT problem

SAT-based exact synthesis

- **Idea:** Solve question *"does there exist a reversible circuit realizing f with r gates?"* as SAT problem
- Try solving question starting from $r = 0$; increase r until solution is found

SAT-based exact synthesis

- **Idea:** Solve question *"does there exist a reversible circuit realizing f with r gates?"* as SAT problem
- Try solving question starting from $r = 0$; increase r until solution is found
- Solution can be extracted from satisfying SAT assignment

SAT-based exact synthesis

- **Idea:** Solve question *"does there exist a reversible circuit realizing f with r gates?"* as SAT problem
- Try solving question starting from $r = 0$; increase r until solution is found
- Solution can be extracted from satisfying SAT assignment
- Only applicable to functions with small number of variables (≈ 7) that require few gates

SAT-based exact synthesis

- **Idea:** Solve question *"does there exist a reversible circuit realizing f with r gates?"* as SAT problem
- Try solving question starting from $r = 0$; increase r until solution is found
- Solution can be extracted from satisfying SAT assignment
- Only applicable to functions with small number of variables (≈ 7) that require few gates

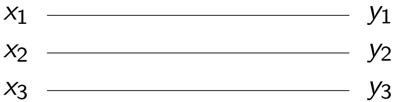
RevKit: `exs`

Reversible synthesis classification

	line opt.	*gate opt.*	**nonreversible func.**	**reversible func.**
functional	✔	✔		SAT-based Enumerative
	✔	✘		Transformation-based Cycle-based Decomposition-based Metaheuristic Greedy
structural	✘	✘	ESOP-based Hierarchical Building block	

Explicit transformation-based synthesis

$x_1x_2x_3$	$y_1y_2y_3$
000	111
001	000
010	110
011	100
100	010
101	001
110	011
111	101



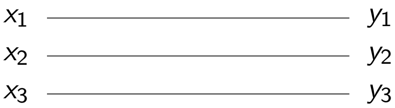
- Apply gates adjust output pattern with input pattern
- By visiting input patterns in numeric order, and by only using positive controlled Toffoli gates, it is ensured that no previous patterns are modified

RevKit: `tbs`

D. Michael Miller, Dmitri Maslov, Gerhard W. Dueck: A transformation based algorithm for reversible logic synthesis, *DAC* **40** (2003), 318–323.

Explicit transformation-based synthesis

$x_1x_2x_3$	$y_1y_2y_3$
000	111
001	000
010	110
011	100
100	010
101	001
110	011
111	101



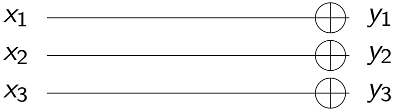
- Apply gates adjust output pattern with input pattern
- By visiting input patterns in numeric order, and by only using positive controlled Toffoli gates, it is ensured that no previous patterns are modified

RevKit: `tbs`

D. Michael Miller, Dmitri Maslov, Gerhard W. Dueck: A transformation based algorithm for reversible logic synthesis, *DAC* **40** (2003), 318–323.

Explicit transformation-based synthesis

$x_1x_2x_3$	$y_1y_2y_3$
000	000
001	111
010	001
011	011
100	101
101	110
110	100
111	010



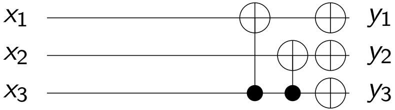
- Apply gates adjust output pattern with input pattern
- By visiting input patterns in numeric order, and by only using positive controlled Toffoli gates, it is ensured that no previous patterns are modified

RevKit: `tbs`

D. Michael Miller, Dmitri Maslov, Gerhard W. Dueck: A transformation based algorithm for reversible logic synthesis, *DAC* **40** (2003), 318–323.

Explicit transformation-based synthesis

$x_1x_2x_3$	$y_1y_2y_3$
000	000
001	001
010	111
011	101
100	011
101	110
110	100
111	010



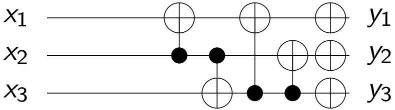
- Apply gates adjust output pattern with input pattern
- By visiting input patterns in numeric order, and by only using positive controlled Toffoli gates, it is ensured that no previous patterns are modified

RevKit: `tbs`

D. Michael Miller, Dmitri Maslov, Gerhard W. Dueck: A transformation based algorithm for reversible logic synthesis, *DAC* **40** (2003), 318–323.

Explicit transformation-based synthesis

$x_1x_2x_3$	$y_1y_2y_3$
000	000
001	001
010	010
011	101
100	110
101	011
110	100
111	111

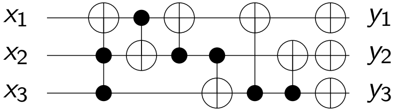


- Apply gates adjust output pattern with input pattern
- By visiting input patterns in numeric order, and by only using positive controlled Toffoli gates, it is ensured that no previous patterns are modified

RevKit: `tbs`

Explicit transformation-based synthesis

$x_1x_2x_3$	$y_1y_2y_3$
000	000
001	001
010	010
011	011
100	100
101	111
110	110
111	101



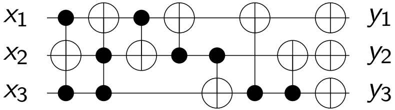
- Apply gates adjust output pattern with input pattern
- By visiting input patterns in numeric order, and by only using positive controlled Toffoli gates, it is ensured that no previous patterns are modified

RevKit: `tbs`

D. Michael Miller, Dmitri Maslov, Gerhard W. Dueck: A transformation based algorithm for reversible logic synthesis, *DAC* **40** (2003), 318–323.

Explicit transformation-based synthesis

$x_1x_2x_3$	$y_1y_2y_3$
000	000
001	001
010	010
011	011
100	100
101	101
110	110
111	111



- Apply gates adjust output pattern with input pattern
- By visiting input patterns in numeric order, and by only using positive controlled Toffoli gates, it is ensured that no previous patterns are modified

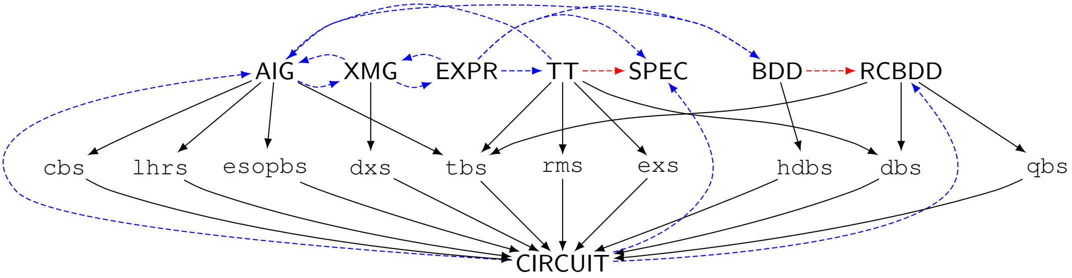
RevKit: `tbs`

D. Michael Miller, Dmitri Maslov, Gerhard W. Dueck: A transformation based algorithm for reversible logic synthesis, *DAC* **40** (2003), 318–323.

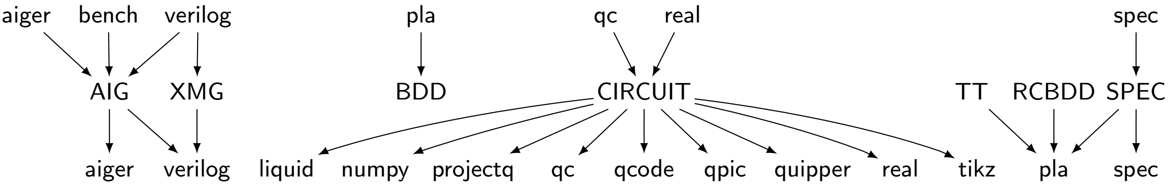
RevKit [Synthesis commands]

	line opt.	*gate opt.*	**nonreversible func.**	**reversible func.**
functional	✔	✔		SAT-based `exs` Enumerative
functional	✔	✘		Transformation-based `tbs`, `rms`, `qbs` Cycle-based `cyclebs` Decomposition-based `dbs` Metaheuristic Greedy
structural	✘	✘	ESOP-based `esopbs` Hierarchical `cbs`, `dxs`, `hdbs`, `lhrs` Building block	

RevKit [Synthesis and formats]



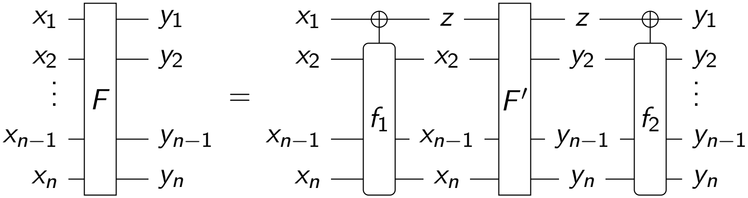
RevKit [File formats]



RevKit [Functional synthesis]

```
revkit> revgen --hwb 6
revkit> tbs
revkit> dbs -n
revkit> store -c
[i] circuits in store:
     0: 6 lines, 141 gates
  *  1: 6 lines, 89 gates
revkit> rec
[i] circuits are equivalent
revkit> reverse
revkit> concat -n
revkit> store -c
[i] circuits in store:
     0: 6 lines, 141 gates
     1: 6 lines, 89 gates
  *  2: 6 lines, 230 gates
revkit> is_identity
[i] circuit represents the identity function
```

Decomposition-based synthesis: idea

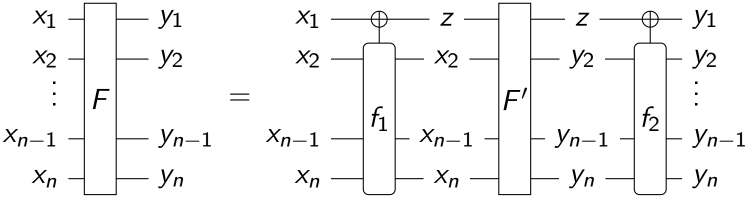


- Every reversible function can be decomposed into three reversible sub-functions $\mathrm{T}_{f_1}(X \setminus \{x_i\}, x_i)$, F', $\mathrm{T}_{f_2}(X \setminus \{x_i\}, x_i)$, where F' is a reversible function that does not change in x_i, for all i

RevKit: `dbs`

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: idea



- Every reversible function can be decomposed into three reversible sub-functions $\mathrm{T}_{f_1}(X \setminus \{x_i\}, x_i)$, F', $\mathrm{T}_{f_2}(X \setminus \{x_i\}, x_i)$, where F' is a reversible function that does not change in x_i, for all i
- Recursive application of this procedure on B in order $i = 1, 2, \ldots, n-1, n$, yields a reversible circuit with $2n-1$ gates, where the targets are aligned in a V-shape

RevKit: `dbs`

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3					y_1	y_2	y_3
0	0	0					0	0	0
0	0	1					0	0	1
0	1	0					0	1	0
0	1	1					1	1	0
1	0	0					1	1	1
1	0	1					0	1	1
1	1	0					1	0	1
1	1	1					1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3							y_1	y_2	y_3
0	0	0	0	0			0	0	0	0	0
0	0	1	0	1			0	1	0	0	1
0	1	0	1	0			1	0	0	1	0
0	1	1	1	1			1	0	1	1	0
1	0	0	0	0			1	1	1	1	1
1	0	1	0	1			1	1	0	1	1
1	1	0	1	0			0	1	1	0	1
1	1	1	1	1			0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3								y_1	y_2	y_3
0	0	0	0	0	0			0	0	0	0	0
0	0	1		0	1			0	1	0	0	1
0	1	0		1	0			1	0	0	1	0
0	1	1		1	1			1	0	1	1	0
1	0	0		0	0			1	1	1	1	1
1	0	1		0	1			1	1	0	1	1
1	1	0		1	0			0	1	1	0	1
1	1	1		1	1			0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0								0	0	0	0	0
0	0	1		0	1								0	1	0	0	1
0	1	0		1	0								1	0	0	1	0
0	1	1		1	1								1	0	1	1	0
1	0	0	1	0	0								1	1	1	1	1
1	0	1		0	1								1	1	0	1	1
1	1	0		1	0								0	1	1	0	1
1	1	1		1	1								0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0								0	0	0	0	0
0	0	1		0	1								0	1	0	0	1
0	1	0		1	0								1	0	0	1	0
0	1	1		1	1								1	0	1	1	0
1	0	0	1	0	0							1	1	1	1	1	1
1	0	1		0	1								1	1	0	1	1
1	1	0		1	0								0	1	1	0	1
1	1	1		1	1								0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3									y_1	y_2	y_3
0	0	0	0	0	0				0	0	0	0	0
0	0	1		0	1				0	1	0	0	1
0	1	0		1	0				1	0	0	1	0
0	1	1		1	1				1	0	1	1	0
1	0	0	1	0	0			1	1	1	1	1	1
1	0	1		0	1			0	1	1	0	1	1
1	1	0		1	0				0	1	1	0	1
1	1	1		1	1				0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3									y_1	y_2	y_3
0	0	0	0	0	0				0	0	0	0	0
0	0	1		0	1				0	1	0	0	1
0	1	0		1	0				1	0	0	1	0
0	1	1		1	1				1	0	1	1	0
1	0	0	1	0	0			1	1	1	1	1	1
1	0	1	0	0	1			0	1	1	0	1	1
1	1	0		1	0				0	1	1	0	1
1	1	1		1	1				0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3									y_1	y_2	y_3
0	0	0	0	0	0				0	0	0	0	0
0	0	1	1	0	1				0	1	0	0	1
0	1	0		1	0				1	0	0	1	0
0	1	1		1	1				1	0	1	1	0
1	0	0	1	0	0			1	1	1	1	1	1
1	0	1	0	0	1			0	1	1	0	1	1
1	1	0		1	0				0	1	1	0	1
1	1	1		1	1				0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0								0	0	0	0	0
0	0	1	1	0	1							1	0	1	0	0	1
0	1	0		1	0								1	0	0	1	0
0	1	1		1	1								1	0	1	1	0
1	0	0	1	0	0							1	1	1	1	1	1
1	0	1	0	0	1							0	1	1	0	1	1
1	1	0		1	0								0	1	1	0	1
1	1	1		1	1								0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0								0	0	0	0	0
0	0	1	1	0	1							1	0	1	0	0	1
0	1	0		1	0								1	0	0	1	0
0	1	1		1	1								1	0	1	1	0
1	0	0	1	0	0							1	1	1	1	1	1
1	0	1	0	0	1							0	1	1	0	1	1
1	1	0		1	0							0	0	1	1	0	1
1	1	1		1	1								0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0								0	0	0	0	0
0	0	1	1	0	1							1	0	1	0	0	1
0	1	0		1	0								1	0	0	1	0
0	1	1		1	1								1	0	1	1	0
1	0	0	1	0	0							1	1	1	1	1	1
1	0	1	0	0	1							0	1	1	0	1	1
1	1	0	0	1	0							0	0	1	1	0	1
1	1	1		1	1								0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0								0	0	0	0	0
0	0	1	1	0	1							1	0	1	0	0	1
0	1	0	1	1	0								1	0	0	1	0
0	1	1		1	1								1	0	1	1	0
1	0	0	1	0	0							1	1	1	1	1	1
1	0	1	0	0	1							0	1	1	0	1	1
1	1	0	0	1	0							0	0	1	1	0	1
1	1	1		1	1								0	0	1	0	0

Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0								0	0	0	0	0
0	0	1	1	0	1							1	0	1	0	0	1
0	1	0	1	1	0							1	1	0	0	1	0
0	1	1		1	1								1	0	1	1	0
1	0	0	1	0	0							1	1	1	1	1	1
1	0	1	0	0	1							0	1	1	0	1	1
1	1	0	0	1	0							0	0	1	1	0	1
1	1	1		1	1								0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3									y_1	y_2	y_3
0	0	0	0	0	0				0	0	0	0	0
0	0	1	1	0	1			1	0	1	0	0	1
0	1	0	1	1	0			1	1	0	0	1	0
0	1	1		1	1			0	1	0	1	1	0
1	0	0	1	0	0			1	1	1	1	1	1
1	0	1	0	0	1			0	1	1	0	1	1
1	1	0	0	1	0			0	0	1	1	0	1
1	1	1		1	1				0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3									y_1	y_2	y_3
0	0	0	0	0	0				0	0	0	0	0
0	0	1	1	0	1			1	0	1	0	0	1
0	1	0	1	1	0			1	1	0	0	1	0
0	1	1	0	1	1			0	1	0	1	1	0
1	0	0	1	0	0			1	1	1	1	1	1
1	0	1	0	0	1			0	1	1	0	1	1
1	1	0	0	1	0			0	0	1	1	0	1
1	1	1		1	1				0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0								0	0	0	0	0
0	0	1	1	0	1							1	0	1	0	0	1
0	1	0	1	1	0							1	1	0	0	1	0
0	1	1	0	1	1							0	1	0	1	1	0
1	0	0	1	0	0							1	1	1	1	1	1
1	0	1	0	0	1							0	1	1	0	1	1
1	1	0	0	1	0							0	0	1	1	0	1
1	1	1	1	1	1								0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3									y_1	y_2	y_3
0	0	0	0	0	0				0	0	0	0	0
0	0	1	1	0	1			1	0	1	0	0	1
0	1	0	1	1	0			1	1	0	0	1	0
0	1	1	0	1	1			0	1	0	1	1	0
1	0	0	1	0	0			1	1	1	1	1	1
1	0	1	0	0	1			0	1	1	0	1	1
1	1	0	0	1	0			0	0	1	1	0	1
1	1	1	1	1	1			1	0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3									y_1	y_2	y_3
0	0	0	0	0	0			0	0	0	0	0	0
0	0	1	1	0	1			1	0	1	0	0	1
0	1	0	1	1	0			1	1	0	0	1	0
0	1	1	0	1	1			0	1	0	1	1	0
1	0	0	1	0	0			1	1	1	1	1	1
1	0	1	0	0	1			0	1	1	0	1	1
1	1	0	0	1	0			0	0	1	1	0	1
1	1	1	1	1	1			1	0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3											y_1	y_2	y_3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	1	1	1	1	1	1	0	1	0	0	1
0	1	0	1	1	0	1	0	1	0	1	1	0	0	1	0
0	1	1	0	1	1	0	1	0	0	0	1	0	1	1	0
1	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1
1	0	1	0	0	1	0	1	0	1	0	1	1	0	1	1
1	1	0	0	1	0	0	0	0	1	0	0	1	1	0	1
1	1	1	1	1	1	1	1	1	0	1	0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0	0	0	0	0		0	0	0	0	0	0	0
0	0	1	1	0	1	1		1	1		1	1	0	1	0	0	1
0	1	0	1	1	0	1		0	1		0	1	1	0	0	1	0
0	1	1	0	1	1	0		1	0		0	0	1	0	1	1	0
1	0	0	1	0	0	1		0	1		1	1	1	1	1	1	1
1	0	1	0	0	1	0		1	0		1	0	1	1	0	1	1
1	1	0	0	1	0	0		0	0		1	0	0	1	1	0	1
1	1	1	1	1	1	1		1	1		0	1	0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0	0	0	0	0		0	0	0	0	0	0	0
0	0	1	1	0	1	1		1	1		1	1	0	1	0	0	1
0	1	0	1	1	0	1		0	1		0	1	1	0	0	1	0
0	1	1	0	1	1	0		1	0		0	0	1	0	1	1	0
1	0	0	1	0	0	1		0	1		1	1	1	1	1	1	1
1	0	1	0	0	1	0		1	0		1	0	1	1	0	1	1
1	1	0	0	1	0	0	1	0	0		1	0	0	1	1	0	1
1	1	1	1	1	1	1		1	1		0	1	0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0	0	0	0	0		0	0	0	0	0	0	0
0	0	1	1	0	1	1		1	1		1	1	0	1	0	0	1
0	1	0	1	1	0	1		0	1		0	1	1	0	0	1	0
0	1	1	0	1	1	0		1	0		0	0	1	0	1	1	0
1	0	0	1	0	0	1		0	1		1	1	1	1	1	1	1
1	0	1	0	0	1	0		1	0		1	0	1	1	0	1	1
1	1	0	0	1	0	0	1	0	0	1	1	0	0	1	1	0	1
1	1	1	1	1	1	1		1	1		0	1	0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0	0	0	0	0		0	0	0	0	0	0	0
0	0	1	1	0	1	1		1	1		1	1	0	1	0	0	1
0	1	0	1	1	0	1		0	1		0	1	1	0	0	1	0
0	1	1	0	1	1	0		1	0		0	0	1	0	1	1	0
1	0	0	1	0	0	1		0	1		1	1	1	1	1	1	1
1	0	1	0	0	1	0		1	0	0	1	0	1	1	0	1	1
1	1	0	0	1	0	0	1	0	0	1	1	0	0	1	1	0	1
1	1	1	1	1	1	1		1	1		0	1	0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0	0	0	0	0		0	0	0	0	0	0	0
0	0	1	1	0	1	1		1	1		1	1	0	1	0	0	1
0	1	0	1	1	0	1		0	1		0	1	1	0	0	1	0
0	1	1	0	1	1	0		1	0		0	0	1	0	1	1	0
1	0	0	1	0	0	1		0	1		1	1	1	1	1	1	1
1	0	1	0	0	1	0	0	1	0	0	1	0	1	1	0	1	1
1	1	0	0	1	0	0	1	0	0	1	1	0	0	1	1	0	1
1	1	1	1	1	1	1		1	1		0	1	0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0	0	0	0	0		0	0	0	0	0	0	0
0	0	1	1	0	1	1		1	1		1	1	0	1	0	0	1
0	1	0	1	1	0	1		0	1		0	1	1	0	0	1	0
0	1	1	0	1	1	0	1	1	0		0	0	1	0	1	1	0
1	0	0	1	0	0	1		0	1		1	1	1	1	1	1	1
1	0	1	0	0	1	0	0	1	0	0	1	0	1	1	0	1	1
1	1	0	0	1	0	0	1	0	0	1	1	0	0	1	1	0	1
1	1	1	1	1	1	1		1	1		0	1	0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0	0	0	0	0		0	0	0	0	0	0	0
0	0	1	1	0	1	1		1	1		1	1	0	1	0	0	1
0	1	0	1	1	0	1		0	1		0	1	1	0	0	1	0
0	1	1	0	1	1	0	1	1	0	1	0	0	1	0	1	1	0
1	0	0	1	0	0	1		0	1		1	1	1	1	1	1	1
1	0	1	0	0	1	0	0	1	0	0	1	0	1	1	0	1	1
1	1	0	0	1	0	0	1	0	0	1	1	0	0	1	1	0	1
1	1	1	1	1	1	1		1	1		0	1	0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	1	1		1	1		1	1	0	1	0	0	1
0	1	0	1	1	0	1		0	1		0	1	1	0	0	1	0
0	1	1	0	1	1	0	1	1	0	1	0	0	1	0	1	1	0
1	0	0	1	0	0	1		0	1		1	1	1	1	1	1	1
1	0	1	0	0	1	0	0	1	0	0	1	0	1	1	0	1	1
1	1	0	0	1	0	0	1	0	0	1	1	0	0	1	1	0	1
1	1	1	1	1	1	1		1	1		0	1	0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	1	1	0	1	1		1	1	0	1	0	0	1
0	1	0	1	1	0	1		0	1		0	1	1	0	0	1	0
0	1	1	0	1	1	0	1	1	0	1	0	0	1	0	1	1	0
1	0	0	1	0	0	1		0	1		1	1	1	1	1	1	1
1	0	1	0	0	1	0	0	1	0	0	1	0	1	1	0	1	1
1	1	0	0	1	0	0	1	0	0	1	1	0	0	1	1	0	1
1	1	1	1	1	1	1		1	1		0	1	0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	1	1	0	1	1		1	1	0	1	0	0	1
0	1	0	1	1	0	1		0	1		0	1	1	0	0	1	0
0	1	1	0	1	1	0	1	1	0	1	0	0	1	0	1	1	0
1	0	0	1	0	0	1		0	1		1	1	1	1	1	1	1
1	0	1	0	0	1	0	0	1	0	0	1	0	1	1	0	1	1
1	1	0	0	1	0	0	1	0	0	1	1	0	0	1	1	0	1
1	1	1	1	1	1	1	1	1	1		0	1	0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	1	1	0	1	1		1	1	0	1	0	0	1
0	1	0	1	1	0	1		0	1		0	1	1	0	0	1	0
0	1	1	0	1	1	0	1	1	0	1	0	0	1	0	1	1	0
1	0	0	1	0	0	1		0	1		1	1	1	1	1	1	1
1	0	1	0	0	1	0	0	1	0	0	1	0	1	1	0	1	1
1	1	0	0	1	0	0	1	0	0	1	1	0	0	1	1	0	1
1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	1	1	0	1	1		1	1	0	1	0	0	1
0	1	0	1	1	0	1		0	1	0	0	1	1	0	0	1	0
0	1	1	0	1	1	0	1	1	0	1	0	0	1	0	1	1	0
1	0	0	1	0	0	1		0	1		1	1	1	1	1	1	1
1	0	1	0	0	1	0	0	1	0	0	1	0	1	1	0	1	1
1	1	0	0	1	0	0	1	0	0	1	1	0	0	1	1	0	1
1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	1	1	0	1	1		1	1	0	1	0	0	1
0	1	0	1	1	0	1	0	0	1	0	0	1	1	0	0	1	0
0	1	1	0	1	1	0	1	1	0	1	0	0	1	0	1	1	0
1	0	0	1	0	0	1		0	1		1	1	1	1	1	1	1
1	0	1	0	0	1	0	0	1	0	0	1	0	1	1	0	1	1
1	1	0	0	1	0	0	1	0	0	1	1	0	0	1	1	0	1
1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	1	1	0	1	1		1	1	0	1	0	0	1
0	1	0	1	1	0	1	0	0	1	0	0	1	1	0	0	1	0
0	1	1	0	1	1	0	1	1	0	1	0	0	1	0	1	1	0
1	0	0	1	0	0	1	1	0	1		1	1	1	1	1	1	1
1	0	1	0	0	1	0	0	1	0	0	1	0	1	1	0	1	1
1	1	0	0	1	0	0	1	0	0	1	1	0	0	1	1	0	1
1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	1	1	0	1	1		1	1	0	1	0	0	1
0	1	0	1	1	0	1	0	0	1	0	0	1	1	0	0	1	0
0	1	1	0	1	1	0	1	1	0	1	0	0	1	0	1	1	0
1	0	0	1	0	0	1	1	0	1	1	1	1	1	1	1	1	1
1	0	1	0	0	1	0	0	1	0	0	1	0	1	1	0	1	1
1	1	0	0	1	0	0	1	0	0	1	1	0	0	1	1	0	1
1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

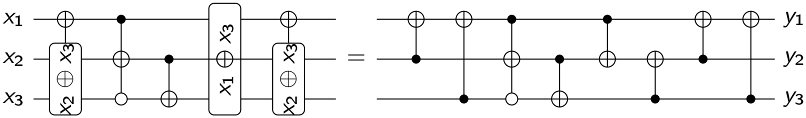
Decomposition-based synthesis: example

x_1	x_2	x_3													y_1	y_2	y_3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	1	1	0	1	1	0	1	1	0	1	0	0	1
0	1	0	1	1	0	1	0	0	1	0	0	1	1	0	0	1	0
0	1	1	0	1	1	0	1	1	0	1	0	0	1	0	1	1	0
1	0	0	1	0	0	1	1	0	1	1	1	1	1	1	1	1	1
1	0	1	0	0	1	0	0	1	0	0	1	0	1	1	0	1	1
1	1	0	0	1	0	0	1	0	0	1	1	0	0	1	1	0	1
1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	1	0	0

A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Decomposition-based synthesis: example

x_1	x_2	x_3														y_1	y_2	y_3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	1	1	0	1	1	0	1	1	0	1	0	0	1
0	1	0	1	1	0	1	0	0	1	0	0	1	1	0	0	1	0
0	1	1	0	1	1	0	1	1	0	1	0	0	1	0	1	1	0
1	0	0	1	0	0	1	1	0	1	1	1	1	1	1	1	1	1
1	0	1	0	0	1	0	0	1	0	0	1	0	1	1	0	1	1
1	1	0	0	1	0	0	1	0	0	1	1	0	0	1	1	0	1
1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	1	0	0



A. De Vos, Y. Van Rentergem: Young subgroups for reversible computers, *Adv. in Math. of Comm.* 2 (2008), 183–200.

Reversible synthesis classification

	line opt.	*gate opt.*	**nonreversible func.**	**reversible func.**
functional	✔	✔		SAT-based Enumerative
functional	✔	✘		Transformation-based Cycle-based Decomposition-based Metaheuristic Greedy
structural	✘	✘	ESOP-based Hierarchical Building block	

ESOP-based synthesis

$$
\begin{aligned}
f(x_1, x_2, x_3, x_4) &= [(x_4 x_3 x_2 x_1)_2 \text{ is prime}] \\
&= \bar{x}_4 \bar{x}_3 x_2 \lor \bar{x}_4 x_3 x_1 \lor x_4 \bar{x}_3 x_2 x_1 \lor x_4 x_3 \bar{x}_2 x_1
\end{aligned}
$$

RevKit: `esopbs`

K. Fazel, M.A. Thornton, and J.E. Rice: ESOP-based Toffoli Gate Cascade Generation, *PACRIM* (2007)

ESOP-based synthesis

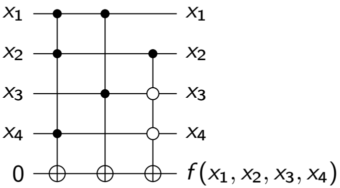
$$
\begin{aligned}
f(x_1, x_2, x_3, x_4) &= [(x_4x_3x_2x_1)_2 \text{ is prime}] \\
&= \bar{x}_4\bar{x}_3x_2 \lor \bar{x}_4x_3x_1 \lor x_4\bar{x}_3x_2x_1 \lor x_4x_3\bar{x}_2x_1 \\
&= x_4x_2x_1 \oplus x_3x_1 \oplus \bar{x}_4\bar{x}_3x_2
\end{aligned}
$$

RevKit: `esopbs`

K. Fazel, M.A. Thornton, and J.E. Rice: ESOP-based Toffoli Gate Cascade Generation, *PACRIM* (2007)

ESOP-based synthesis

$$f(x_1, x_2, x_3, x_4) = [(x_4x_3x_2x_1)_2 \text{ is prime}]$$
$$= \bar{x}_4\bar{x}_3x_2 \lor \bar{x}_4x_3x_1 \lor x_4\bar{x}_3x_2x_1 \lor x_4x_3\bar{x}_2x_1$$
$$= x_4x_2x_1 \oplus x_3x_1 \oplus \bar{x}_4\bar{x}_3x_2$$

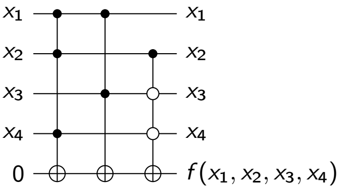


RevKit: `esopbs`

K. Fazel, M.A. Thornton, and J.E. Rice: ESOP-based Toffoli Gate Cascade Generation, *PACRIM* (2007)

ESOP-based synthesis

$$f(x_1,x_2,x_3,x_4) = [(x_4x_3x_2x_1)_2 \text{ is prime}]$$
$$= \bar{x}_4\bar{x}_3x_2 \lor \bar{x}_4x_3x_1 \lor x_4\bar{x}_3x_2x_1 \lor x_4x_3\bar{x}_2x_1$$
$$= x_4x_2x_1 \oplus x_3x_1 \oplus \bar{x}_4\bar{x}_3x_2$$

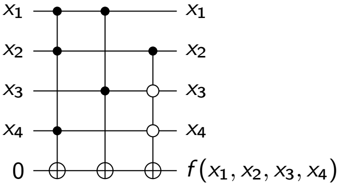


$$f_1 = [3 \mid (x_4x_3x_2x_1)_2]$$
$$= \bar{x}_1\bar{x}_2x_3 \oplus \bar{x}_1\bar{x}_4 \oplus x_1\bar{x}_3x_4 \oplus$$
$$x_1x_2x_4 \oplus x_2\bar{x}_3\bar{x}_4$$
$$f_2 = [(x_4x_3x_2x_1)_2 \text{ is prime}]$$

RevKit: `esopbs`

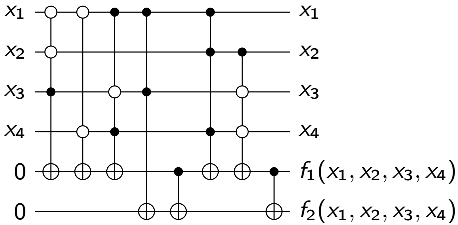
ESOP-based synthesis

$$f(x_1,x_2,x_3,x_4) = [(x_4x_3x_2x_1)_2 \text{ is prime}]$$
$$= \bar{x}_4\bar{x}_3x_2 \vee \bar{x}_4x_3x_1 \vee x_4\bar{x}_3x_2x_1 \vee x_4x_3\bar{x}_2x_1$$
$$= x_4x_2x_1 \oplus x_3x_1 \oplus \bar{x}_4\bar{x}_3x_2$$



RevKit: `esopbs`

$$f_1 = [3 \mid (x_4x_3x_2x_1)_2]$$
$$= \bar{x}_1\bar{x}_2x_3 \oplus \bar{x}_1\bar{x}_4 \oplus x_1\bar{x}_3x_4 \oplus$$
$$x_1x_2x_4 \oplus x_2\bar{x}_3\bar{x}_4$$
$$f_2 = [(x_4x_3x_2x_1)_2 \text{ is prime}]$$



RevKit [General commands]

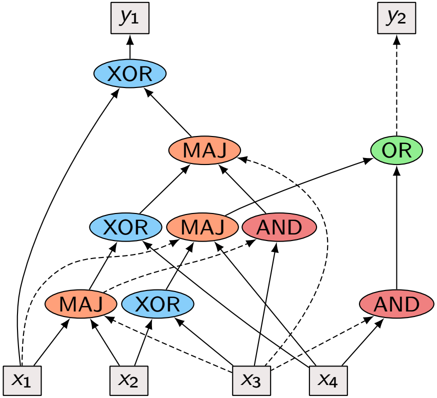
- **alias** — creates an alias
- **convert** — converts one store element into another
- **current** — changes current store element
- **help** — shows all commands
- **print** — prints current store element as ASCII
- **ps** — prints statistics about current store element
- **quit** — quits RevKit
- **set** — sets global (settings) variable
- **show** — generates visual representation of current store element (as DOT file)
- **store** — interact with the store

Reversible synthesis classification

	line opt.	*gate opt.*	**nonreversible func.**	**reversible func.**
functional	✔	✔		SAT-based Enumerative
functional	✔	✘		Transformation-based Cycle-based Decomposition-based Metaheuristic Greedy
structural	✘	✘	ESOP-based Hierarchical Building block	

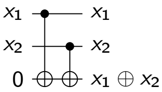
XMG-based synthesis

- XMG consists of **XOR** gates with 2 inputs and **MAJ** gates with 3 inputs
- MAJ gates with constant input can represent **AND** and **OR** gates
- Edges can be complemented (dashed lines in graph)

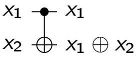


RevKit: `dxs`

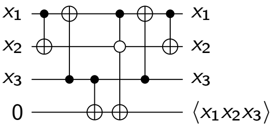
XMG-based synthesis



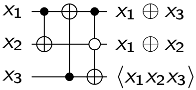
XOR



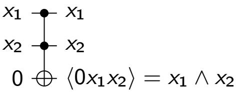
XOR (in-place)



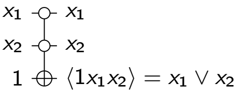
MAJ



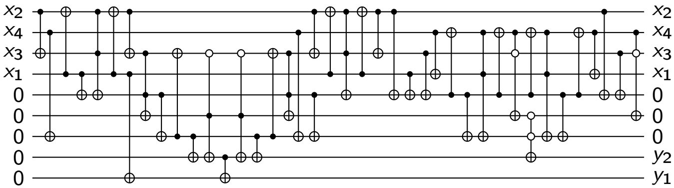
MAJ (in-place)



AND



OR



LUT-based hierarchical reversible synthesis

Goal: Automatically synthesizing large Boolean functions into Clifford+T networks of reasonable quality (qubits and T-count)

Soeken, Roetteler, Wiebe, De Micheli: Hierarchical reversible logic synthesis using LUTs, *DAC* (2017), arXiv:1706.02721

LUT-based hierarchical reversible synthesis

Goal: Automatically synthesizing large Boolean functions into Clifford+T networks of reasonable quality (qubits and T-count)

Algorithm: LUT-based hierarchical reversible synthesis (LHRS)

1. Represent input function as classical logic network and optimize it

RevKit: `lhrs`

Soeken, Roetteler, Wiebe, De Micheli: Hierarchical reversible logic synthesis using LUTs, *DAC* (2017), arXiv:1706.02721

LUT-based hierarchical reversible synthesis

Goal: Automatically synthesizing large Boolean functions into Clifford+T networks of reasonable quality (qubits and T-count)

Algorithm: LUT-based hierarchical reversible synthesis (LHRS)

1. Represent input function as classical logic network and optimize it
2. Map network into k-LUT network

RevKit: `lhrs`

Soeken, Roetteler, Wiebe, De Micheli: Hierarchical reversible logic synthesis using LUTs, *DAC* (2017), arXiv:1706.02721

LUT-based hierarchical reversible synthesis

Goal: Automatically synthesizing large Boolean functions into Clifford+T networks of reasonable quality (qubits and T-count)

Algorithm: LUT-based hierarchical reversible synthesis (LHRS)

1. Represent input function as classical logic network and optimize it
2. Map network into k-LUT network
3. Translate k-LUT network into reversible network with k-input single-target gates

RevKit: `lhrs`

Soeken, Roetteler, Wiebe, De Micheli: Hierarchical reversible logic synthesis using LUTs, *DAC* (2017), arXiv:1706.02721

LUT-based hierarchical reversible synthesis

Goal: Automatically synthesizing large Boolean functions into Clifford+T networks of reasonable quality (qubits and T-count)

Algorithm: LUT-based hierarchical reversible synthesis (LHRS)

1. Represent input function as classical logic network and optimize it
2. Map network into k-LUT network
3. Translate k-LUT network into reversible network with k-input single-target gates
4. Map single-target gates into Clifford+T networks

RevKit: `lhrs`

Soeken, Roetteler, Wiebe, De Micheli: Hierarchical reversible logic synthesis using LUTs, *DAC* (2017), arXiv:1706.02721

LUT-based hierarchical reversible synthesis

Goal: Automatically synthesizing large Boolean functions into Clifford+T networks of reasonable quality (qubits and T-count)

Algorithm: LUT-based hierarchical reversible synthesis (LHRS)

conv. alg.

1. Represent input function as classical logic network and optimize it
2. Map network into k-LUT network
3. Translate k-LUT network into reversible network with k-input single-target gates
4. Map single-target gates into Clifford+T networks

RevKit: `lhrs`

Soeken, Roetteler, Wiebe, De Micheli: Hierarchical reversible logic synthesis using LUTs, *DAC* (2017), arXiv:1706.02721

LUT-based hierarchical reversible synthesis

Goal: Automatically synthesizing large Boolean functions into Clifford+T networks of reasonable quality (qubits and T-count)

Algorithm: LUT-based hierarchical reversible synthesis (LHRS)

conv. alg.

1. Represent input function as classical logic network and optimize it
2. Map network into k-LUT network

new alg.

3. Translate k-LUT network into reversible network with k-input single-target gates
4. Map single-target gates into Clifford+T networks

RevKit: `lhrs`

Soeken, Roetteler, Wiebe, De Micheli: Hierarchical reversible logic synthesis using LUTs, *DAC* (2017), arXiv:1706.02721

LUT-based hierarchical reversible synthesis

Goal: Automatically synthesizing large Boolean functions into Clifford+T networks of reasonable quality (qubits and T-count)

Algorithm: LUT-based hierarchical reversible synthesis (LHRS)

conv. alg.

new alg.

1. Represent input function as classical logic network and optimize it
2. Map network into k-LUT network
3. Translate k-LUT network into reversible network with k-input single-target gates
4. Map single-target gates into Clifford+T networks

affects #qubits

RevKit: `lhrs`

Soeken, Roetteler, Wiebe, De Micheli: Hierarchical reversible logic synthesis using LUTs, *DAC* (2017), arXiv:1706.02721

LUT-based hierarchical reversible synthesis

Goal: Automatically synthesizing large Boolean functions into Clifford+T networks of reasonable quality (qubits and T-count)

Algorithm: LUT-based hierarchical reversible synthesis (LHRS)

conv. alg.

1. Represent input function as classical logic network and optimize it
2. Map network into k-LUT network

new alg.

3. Translate k-LUT network into reversible network with k-input single-target gates
4. Map single-target gates into Clifford+T networks

affects #qubits

affects # T gates

RevKit: `lhrs`

Soeken, Roetteler, Wiebe, De Micheli: Hierarchical reversible logic synthesis using LUTs, *DAC* (2017), arXiv:1706.02721

 |

LUT mapping

- Realizing a logic function or logic circuit in terms of a k-LUT logic network
- A k-LUT is any Boolean function with at most k inputs
- One of the most effective methods used in logic synthesis

Soeken, Roetteler, Wiebe, De Micheli: Hierarchical reversible logic synthesis using LUTs, *DAC* (2017), arXiv:1706.02721

LUT mapping

- Realizing a logic function or logic circuit in terms of a k-LUT logic network
- A k-LUT is any Boolean function with at most k inputs
- One of the most effective methods used in logic synthesis
- Typical objective functions are size (number of LUTs) and depth (longest path from inputs to outputs)
- Open source software ABC can generate industrial-scale mappings

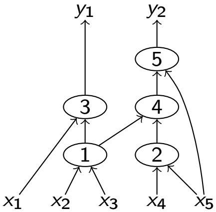
Soeken, Roetteler, Wiebe, De Micheli: Hierarchical reversible logic synthesis using LUTs, *DAC* (2017), arXiv:1706.02721

LUT mapping

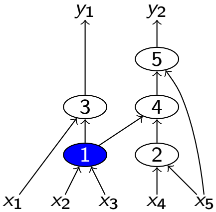
- Realizing a logic function or logic circuit in terms of a k-LUT logic network
- A k-LUT is any Boolean function with at most k inputs
- One of the most effective methods used in logic synthesis
- Typical objective functions are size (number of LUTs) and depth (longest path from inputs to outputs)
- Open source software ABC can generate industrial-scale mappings
- Can be used as technology mapper for FPGAs (e.g., when $k \leq 7$)

Soeken, Roetteler, Wiebe, De Micheli: Hierarchical reversible logic synthesis using LUTs, *DAC* (2017), arXiv:1706.02721

k-LUT network to reversible network

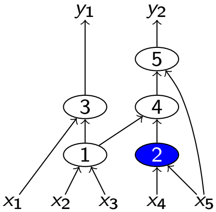


k-LUT network to reversible network



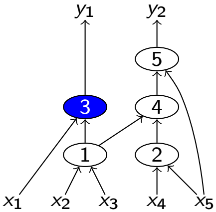
- ! k-LUT corresponds to k-controlled single-target gate

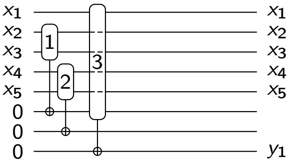
k-LUT network to reversible network



! k-LUT corresponds to k-controlled single-target gate

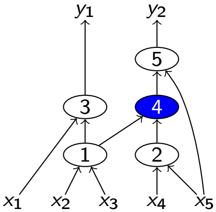
k-LUT network to reversible network

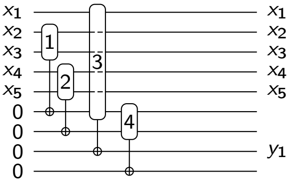




! k-LUT corresponds to k-controlled single-target gate

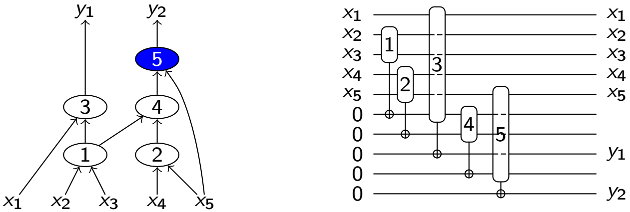
k-LUT network to reversible network





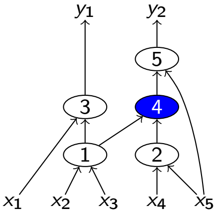
! k-LUT corresponds to k-controlled single-target gate

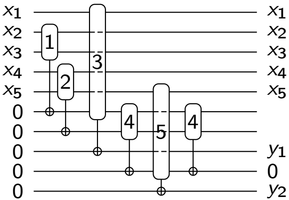
k-LUT network to reversible network



! k-LUT corresponds to k-controlled single-target gate

k-LUT network to reversible network

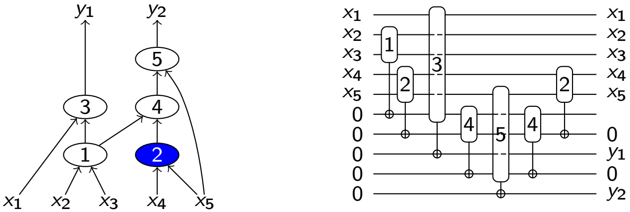




! k-LUT corresponds to k-controlled single-target gate

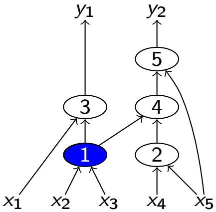
- non-output LUTs need to be uncomputed

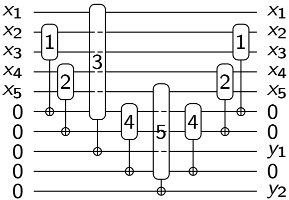
k-LUT network to reversible network



- ! k-LUT corresponds to k-controlled single-target gate
- non-output LUTs need to be uncomputed

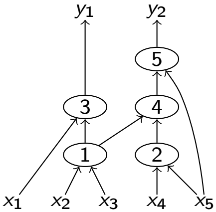
k-LUT network to reversible network

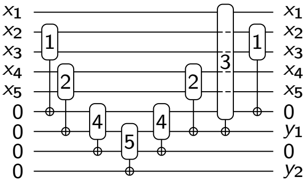




- ! k-LUT corresponds to k-controlled single-target gate
- non-output LUTs need to be uncomputed

k-LUT network to reversible network



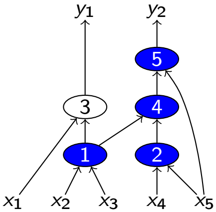


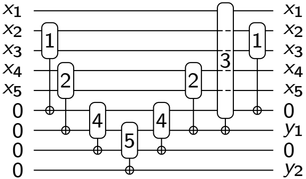
! k-LUT corresponds to k-controlled single-target gate

- non-output LUTs need to be uncomputed
- order of LUT traversal determines number of ancillas

Soeken, Roetteler, Wiebe, De Micheli: Hierarchical reversible logic synthesis using LUTs, *DAC* (2017), arXiv:1706.02721

k-LUT network to reversible network

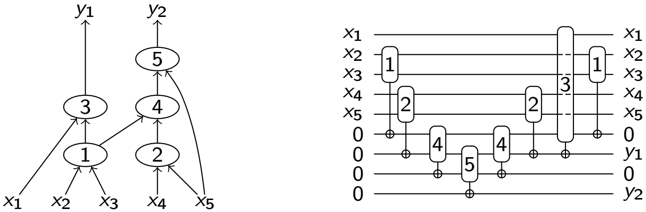




- **!** k-LUT corresponds to k-controlled single-target gate
- non-output LUTs need to be uncomputed
- order of LUT traversal determines number of ancillas
- maximum output cone determines minimum number of ancillas

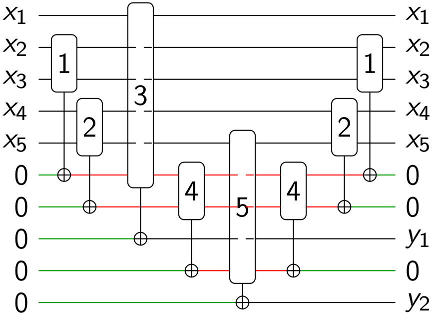
Soeken, Roetteler, Wiebe, De Micheli: Hierarchical reversible logic synthesis using LUTs, *DAC* (2017), arXiv:1706.02721

k-LUT network to reversible network



- ! k-LUT corresponds to k-controlled single-target gate
- ▸ non-output LUTs need to be uncomputed
- ▸ order of LUT traversal determines number of ancillas
- ▸ maximum output cone determines minimum number of ancillas
- ☺ fast mapping that generates a fixed-space skeleton for subnetwork synthesis

Single-target gate LUT mapping



- Mapping problem: Given a single-target gate $\mathrm{T}_f(X, x_t)$ (with control function f, control lines X, and target line x_t), a set of clean ancillas X_c, and a set of dirty ancillas X_d, find the best mapping into a Clifford+T network, such that all ancillas are restored to their original value.

Single-target gate mapping algorithms

- Direct

Single-target gate mapping algorithms

- Direct
  - Map each control function using ESOP based synthesis

Single-target gate mapping algorithms

- Direct
  - Map each control function using ESOP based synthesis
  - Does not use ancillae

Single-target gate mapping algorithms

- Direct
  - Map each control function using ESOP based synthesis
  - Does not use ancillae
- LUT-based

Single-target gate mapping algorithms

- Direct
  - Map each control function using ESOP based synthesis
  - Does not use ancillae
- LUT-based
  - Map control function into smaller LUT network

Soeken, Roetteler, Wiebe, De Micheli: Hierarchical reversible logic synthesis using LUTs, *DAC* (2017), arXiv:1706.02721

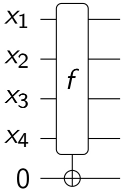
Single-target gate mapping algorithms

- Direct
  - Map each control function using ESOP based synthesis
  - Does not use ancillae
- LUT-based
  - Map control function into smaller LUT network
  - Map small LUTs into pre-computed optimum quantum circuits

Soeken, Roetteler, Wiebe, De Micheli: Hierarchical reversible logic synthesis using LUTs, *DAC* (2017), arXiv:1706.02721

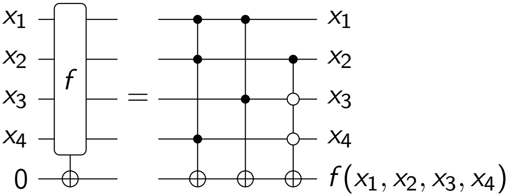
Direct mapping

$$f(x_1, x_2, x_3, x_4) = [(x_4x_3x_2x_1)_2 \text{ is prime}]$$
$$= \bar{x}_4\bar{x}_3x_2 \vee \bar{x}_4x_3x_1 \vee x_4\bar{x}_3x_2x_1 \vee x_4x_3\bar{x}_2x_1$$



Direct mapping

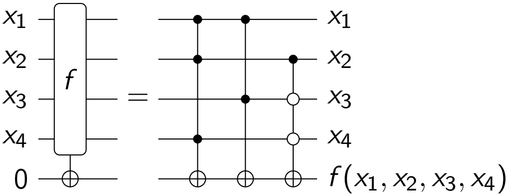
$$\begin{aligned} f(x_1, x_2, x_3, x_4) &= [(x_4x_3x_2x_1)_2 \text{ is prime}] \\ &= \bar{x}_4\bar{x}_3x_2 \lor \bar{x}_4x_3x_1 \lor x_4\bar{x}_3x_2x_1 \lor x_4x_3\bar{x}_2x_1 \\ &= x_4x_2x_1 \oplus x_3x_1 \oplus \bar{x}_4\bar{x}_3x_2 \end{aligned}$$



Soeken, Roetteler, Wiebe, De Micheli: Hierarchical reversible logic synthesis using LUTs, *DAC* (2017), arXiv:1706.02721

Direct mapping

$$
\begin{aligned}
f(x_1, x_2, x_3, x_4) &= [(x_4 x_3 x_2 x_1)_2 \text{ is prime}] \\
&= \bar{x}_4 \bar{x}_3 x_2 \lor \bar{x}_4 x_3 x_1 \lor x_4 \bar{x}_3 x_2 x_1 \lor x_4 x_3 \bar{x}_2 x_1 \\
&= x_4 x_2 x_1 \oplus x_3 x_1 \oplus \bar{x}_4 \bar{x}_3 x_2
\end{aligned}
$$

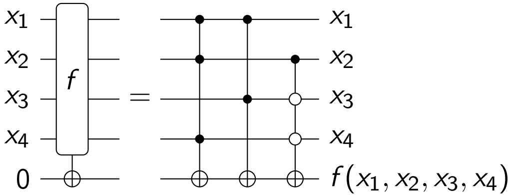


- Each multiple-controlled Toffoli gate is mapped to Clifford+T

Soeken, Roetteler, Wiebe, De Micheli: Hierarchical reversible logic synthesis using LUTs, *DAC* (2017), arXiv:1706.02721

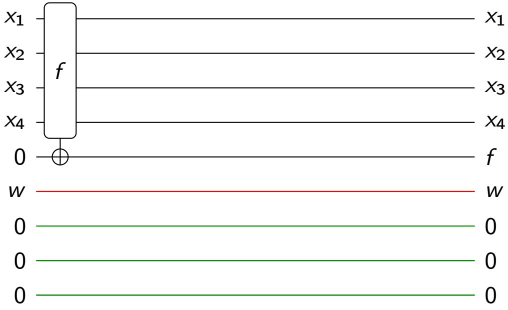
Direct mapping

$$\begin{aligned} f(x_1, x_2, x_3, x_4) &= [(x_4x_3x_2x_1)_2 \text{ is prime}] \\ &= \bar{x}_4\bar{x}_3x_2 \vee \bar{x}_4x_3x_1 \vee x_4\bar{x}_3x_2x_1 \vee x_4x_3\bar{x}_2x_1 \\ &= x_4x_2x_1 \oplus x_3x_1 \oplus \bar{x}_4\bar{x}_3x_2 \end{aligned}$$

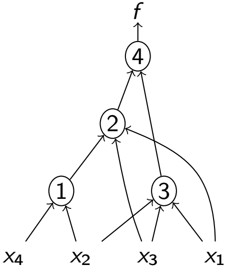


- Each multiple-controlled Toffoli gate is mapped to Clifford+T
- ☹ ESOP minimization tools (e.g., exorcism) optimize for cube count

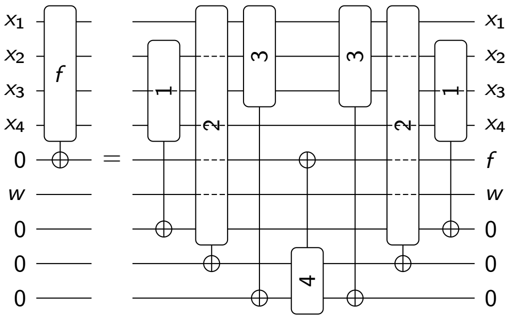
LUT-based single-target gate mapping

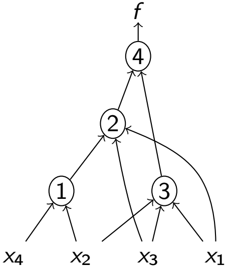


LUT-based single-target gate mapping

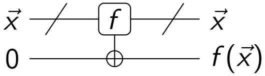


LUT-based single-target gate mapping

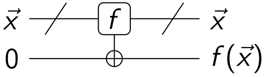


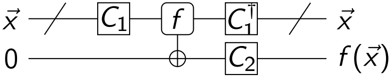


Affine transformations



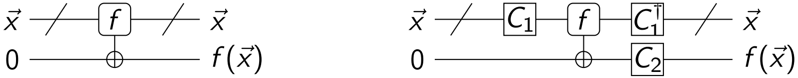
Affine transformations





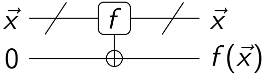
- Let C_1 and C_2 be circuits that only consist of NOT and CNOT gates

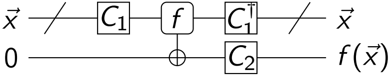
Affine transformations



- Let C_1 and C_2 be circuits that only consist of NOT and CNOT gates
- Both circuits have the same number of T gates

Affine transformations

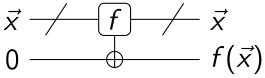


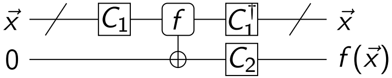


- Let C_1 and C_2 be circuits that only consist of NOT and CNOT gates
- Both circuits have the same number of T gates

☺ Instead of 65 536 4-input functions we only need to consider 18

Affine transformations

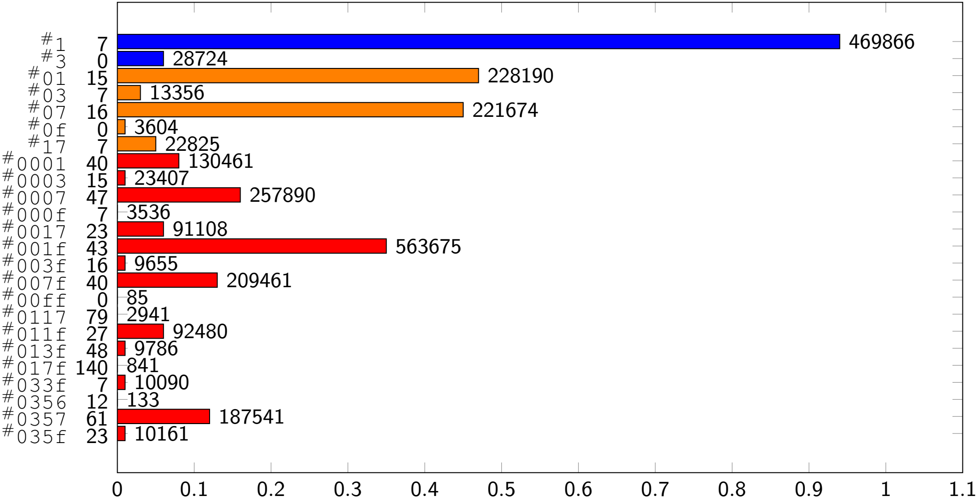




- Let C_1 and C_2 be circuits that only consist of NOT and CNOT gates
- Both circuits have the same number of T gates
- ☺ Instead of 65 536 4-input functions we only need to consider 18
- ☺ Instead of 4 294 967 296 5-input functions we only need to consider 206

Soeken, Roetteler, Wiebe, De Micheli: Hierarchical reversible logic synthesis using LUTs, *DAC* (2017), arXiv:1706.02721

Circuits for affine equivalence classes



quantumlib.stationq.com

The LHRS ecosystem

arxiv.org/abs/1706.02721

LHRS

- Mapping into LUTs
- Aligning LUTs as single-target gates
- Mapping single-target gates

Soeken, Roetteler, Wiebe, De Micheli: Hierarchical reversible logic synthesis using LUTs, *DAC* (2017), arXiv:1706.02721

The LHRS ecosystem

arxiv.org/abs/1706.02721

</> program

```
let gaussian a b c d x =
  let den = (x - b) * (x - b)
  let nom = -2.0f * c * c
  a * exp (den / nom)
```

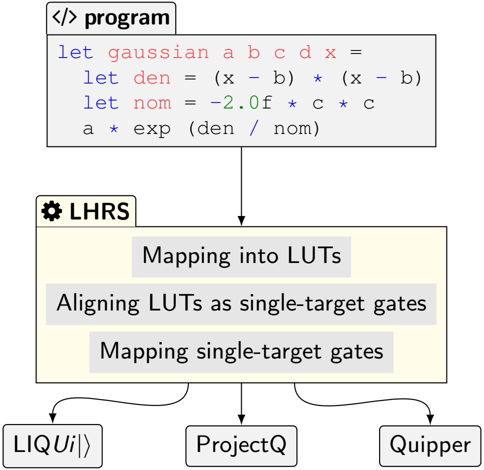
LHRS

- Mapping into LUTs
- Aligning LUTs as single-target gates
- Mapping single-target gates

Soeken, Roetteler, Wiebe, De Micheli: Hierarchical reversible logic synthesis using LUTs, *DAC* (2017), arXiv:1706.02721

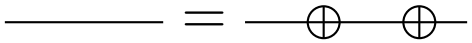
The LHRS ecosystem

arxiv.org/abs/1706.02721



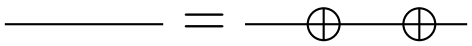
Soeken, Roetteler, Wiebe, De Micheli: Hierarchical reversible logic synthesis using LUTs, *DAC* (2017), arXiv:1706.02721

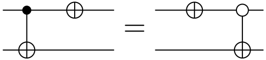
Circuit rewriting

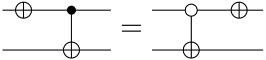


M. Soeken, M.K. Thomsen: White dots *do* matter: rewriting reversible logic circuits, *RC* **5**, (2013), 196–208

Circuit rewriting

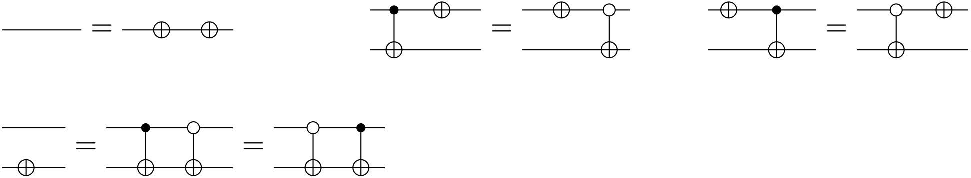






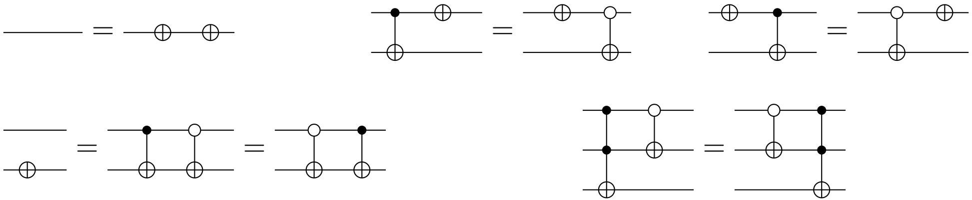
M. Soeken, M.K. Thomsen: White dots *do* matter: rewriting reversible logic circuits, *RC* 5, (2013), 196–208

Circuit rewriting



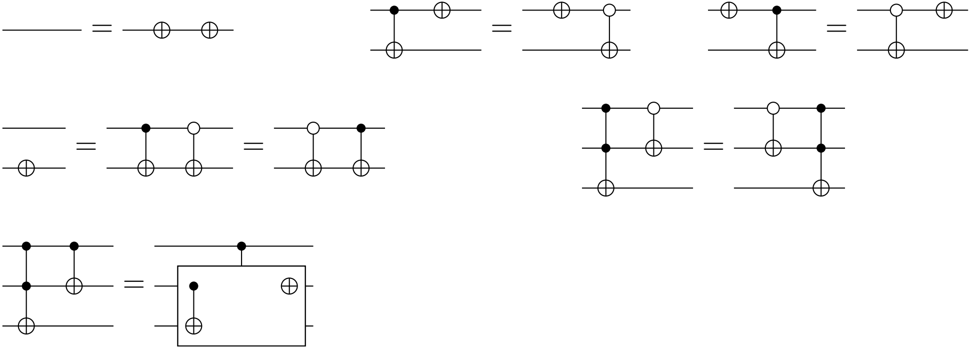
M. Soeken, M.K. Thomsen: White dots *do* matter: rewriting reversible logic circuits, *RC* 5, (2013), 196–208

Circuit rewriting



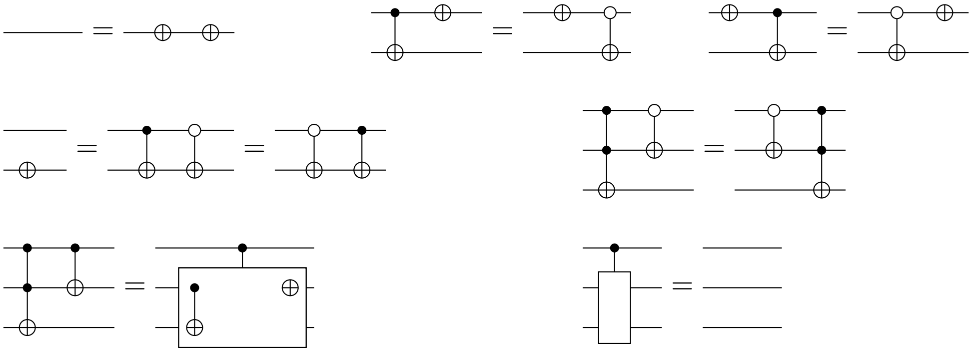
M. Soeken, M.K. Thomsen: White dots *do* matter: rewriting reversible logic circuits, *RC* 5, (2013), 196–208

Circuit rewriting



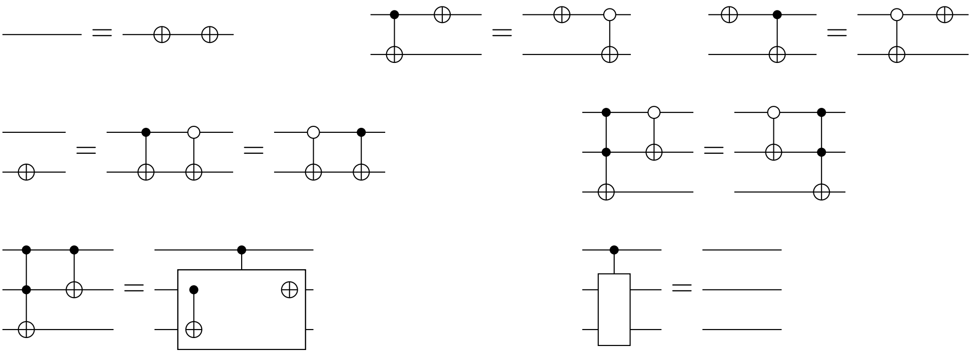
M. Soeken, M.K. Thomsen: White dots *do* matter: rewriting reversible logic circuits, *RC* 5, (2013), 196–208

Circuit rewriting



M. Soeken, M.K. Thomsen: White dots *do* matter: rewriting reversible logic circuits, *RC* 5, (2013), 196–208

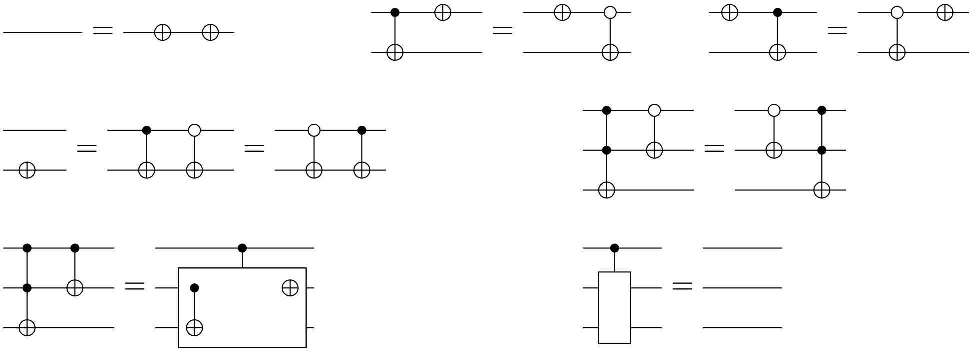
Circuit rewriting



? Open problem: These six rules (plus SWAP rule) are complete, i.e., one can rewrite any circuit realizing some function into any other circuit realizing the same function

M. Soeken, M.K. Thomsen: White dots *do* matter: rewriting reversible logic circuits, *RC* 5, (2013), 196–208

Circuit rewriting



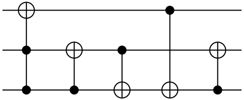
? **Open problem:** These six rules (plus SWAP rule) are complete, i.e., one can rewrite any circuit realizing some function into any other circuit realizing the same function

- Rule set has been extended to consider ancillae

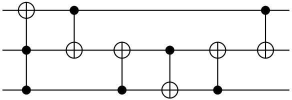
M. Soeken, M.K. Thomsen: White dots *do* matter: rewriting reversible logic circuits, *RC* 5, (2013), 196–208

Circuit rewriting: example

Circuit G_1



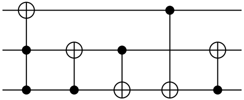
Circuit G_2



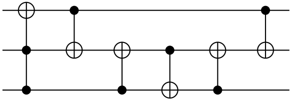
M. Soeken, M.K. Thomsen: White dots *do* matter: rewriting reversible logic circuits, *RC* 5, (2013), 196–208

Circuit rewriting: example

Circuit G_1



Circuit G_2

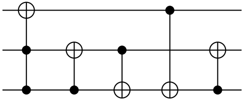


- Can be used for equivalence checking to check $G_1 \equiv G_2$
- Construct circuit $G = G_2^{-1} \circ G_1$
- Rewrite G to identity

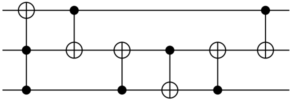
M. Soeken, M.K. Thomsen: White dots *do* matter: rewriting reversible logic circuits, *RC* 5, (2013), 196–208

Circuit rewriting: example

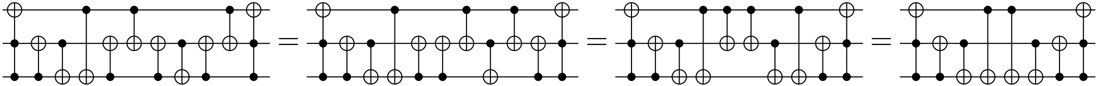
Circuit G_1



Circuit G_2

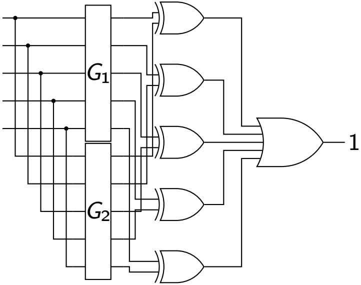


- Can be used for equivalence checking to check $G_1 \equiv G_2$
- Construct circuit $G = G_2^{-1} \circ G_1$
- Rewrite G to identity



Equivalence checking of reversible circuits

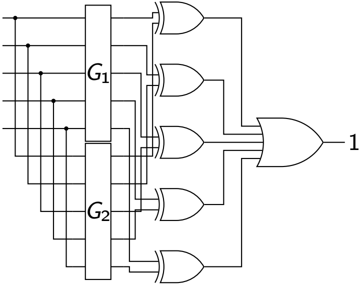
Conventional approach



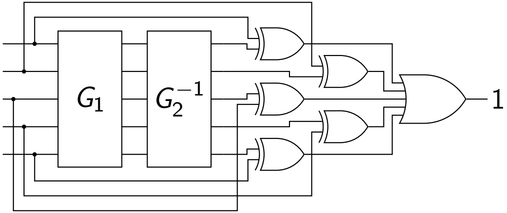
Exploiting reversibility

Equivalence checking of reversible circuits

Conventional approach



Exploiting reversibility

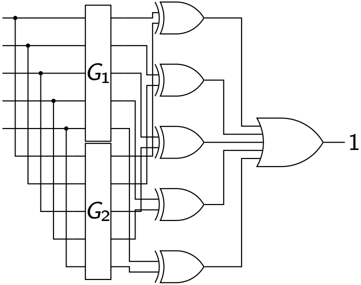


RevKit: `rec`

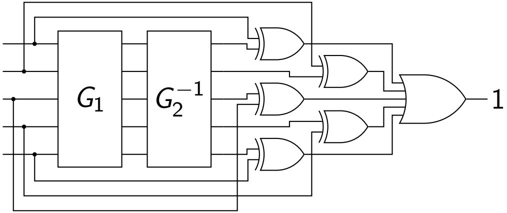
L.G. Amarù, P.-E. Gaillardon, R. Wille, and G. De Micheli: *DATE* 19 (2016)

Equivalence checking of reversible circuits

Conventional approach



Exploiting reversibility



RevKit: `rec`

- Circuit is translated into a SAT formula and solved with a SAT solver
- A satisfying solution is witnessing a counter-example
- Solvers with support for XOR clauses allow for more natural encoding and better runtimes

Introduction and background

Reversible logic synthesis

Introduction into RevKit

Reversible logic synthesis and RevKit

Mathias Soeken

Integrated Systems Laboratory, EPFL, Switzerland

mathias.soeken@epfl.ch msoeken.github.io msoeken/cirkit