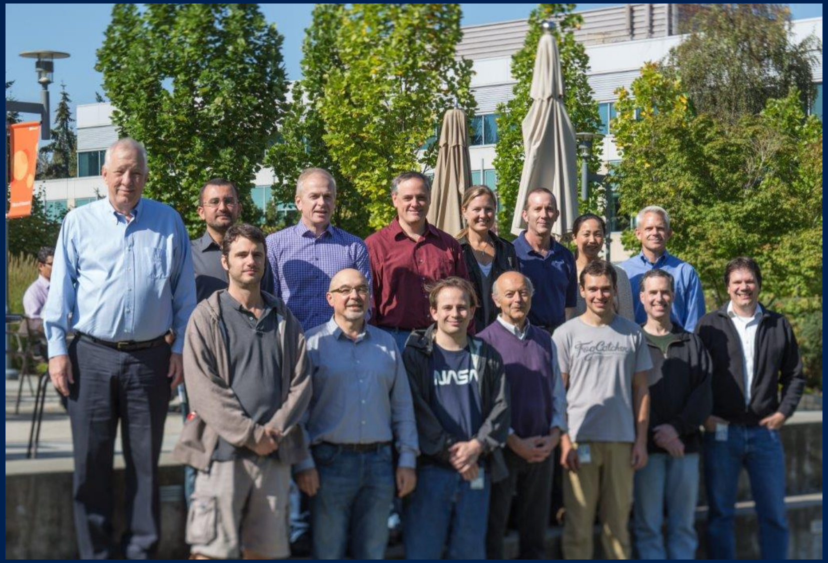
Reversible circuit compilation with space constraints

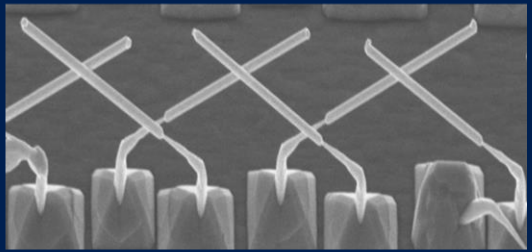
Martin Roetteler
Quantum Architectures and Computation Group (QuArC)
Microsoft Research

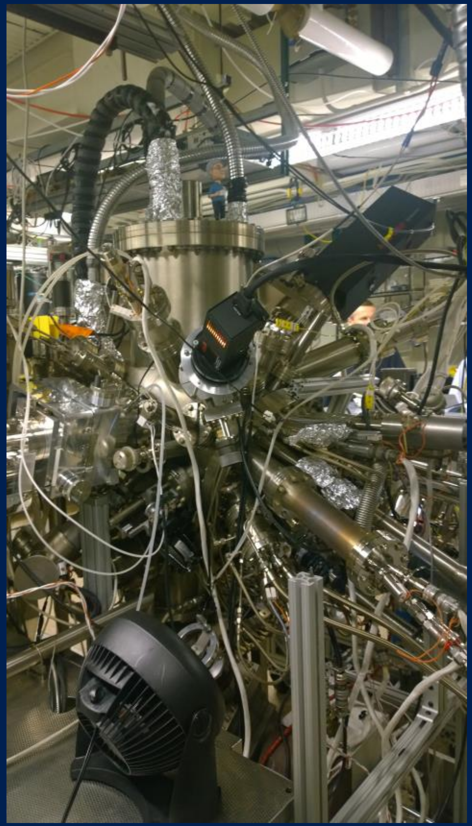
Based on joint work with Matt Amy, Alex Parent, and Krysta M. Svore:
arXiv:1510.00377 arxiv:1603.01635

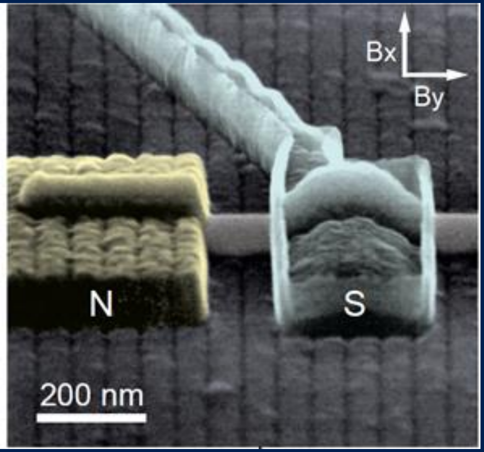
QPL 2016
Glasgow, June 9, 2016

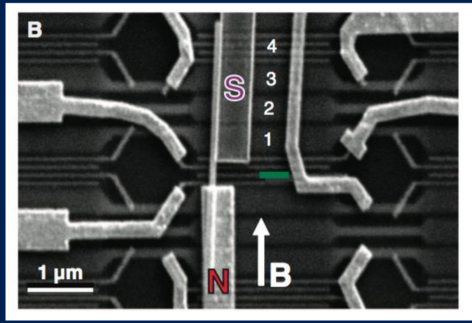
Microsoft QuArC and StationQ











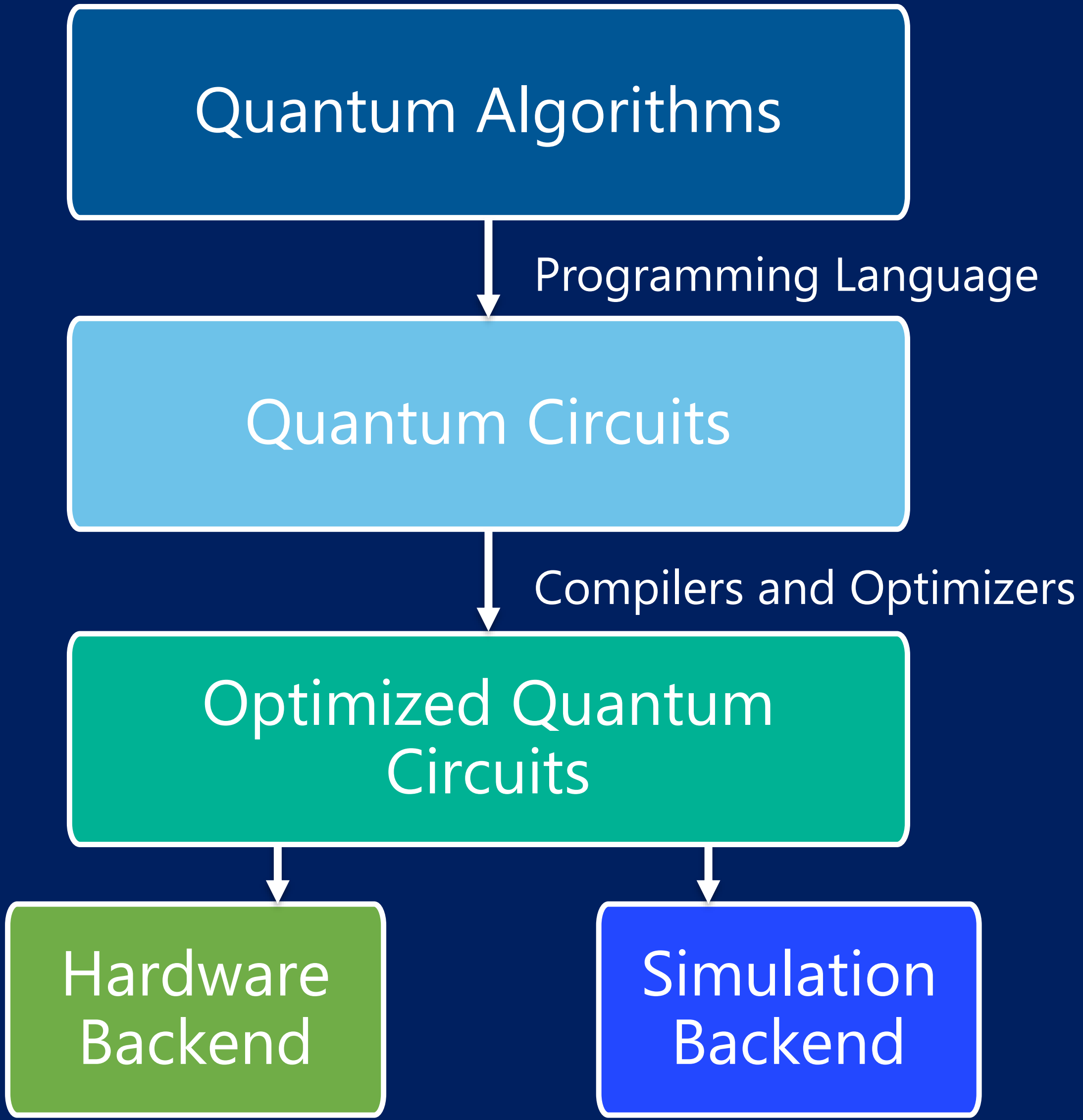
Quantum programming in LIQ$Ui|\rangle$

LIQ$Ui|\rangle$ goals

- Simulation:
  - High enough level language to easily implement large quantum algorithms
  - Allow as large a simulation on classical computers as possible
  - Support abstraction and visualization to help the user
  - Implement as an extensible platform so users can tailor to their own requirements

- Compilation:
  - Multi-level analysis of circuits to allow many types of optimization
  - Circuit re-writing for specific needs (e.g., different gate sets, noise modeling)
  - Compilation into real target architectures

A software architecture for quantum computing



- Goal: automatically translate quantum algorithm to executable code for a quantum computer
- Increases speed of innovation
  - Rapid development of quantum algorithms
  - Efficient testing of architectural designs
  - Flexible for the future

The LIQ$Ui|\rangle$ platform

Wecker and Svore, 2014

The LIQUi|⟩ simulation platform

- We chos… …um algorithms
  - F# is als…
- Optimize…
  - Paralleli…
  - Many hi… …s growing a complex… …n
  - A CHP-… …ns that don't require full circu…
- Public re…
  - Restricte…
  - No software restrictions on the stabilizer simulator

LIQUi|>: A Software Design Architecture and Domain-Specific Language for Quantum Computing. Dave Wecker, Krysta M. Svore

Languages, compilers, and computer-aided design tools will be essential for scalable quantum computing, which promises an exponential leap in our ability to execute complex tasks. LIQUi|> is a modular software architecture designed to control quantum hardware. It enables easy programming, compilation, and simulation of quantum algorithms and circuits, and is independent of a specific quantum architecture. LIQUi|> contains an embedded, domain-specific language designed for programming quantum algorithms, with F# as the host language. It also allows the extraction of a circuit data structure that can be used for optimization, rendering, or translation. The circuit can also be exported to external hardware and software environments. Two different simulation environments are available to the user which allow a trade-off between number of qubits and class of operations. LIQUi|> has been implemented on a wide range of runtimes as back-ends with a single user front-end. We describe the significant components of the design architecture and how to express any given quantum algorithm.

Paper: http://arxiv.org/abs/1402.4467

Software: http://stationq.github.io/Liquid

First coding challenge just ended

Interested in delving into quantum chemistry, linear algebra, teleportation, and much more? Students entered the Microsoft Quantum Challenge to see how far they could go! From around the world students investigated and solved problems facing the quantum universe using Microsoft's simulator, LIQ*Ui*|>.

They won big prizes, and the opportunity to visit Microsoft Research and maybe gain an internship.

Winners

We are delighted to announce the winners of the Challenge. Interest over the past three months came from all round the world. The judging panel was impressed by all the entries. The following were chosen to receive prizes. Congratulations to the winners!

Each of the winners used the simulator for Language-Integrated Quantum Operations: LIQ*Ui*|> from Microsoft Research. Read more on our Blog.

Thien Nguyen
Research School of Engineering, Australian National University, Canberra, Australia
Grand Prize - $5,000
Entry: Simulating Dynamical Input-Output Quantum Systems with LIQ*Ui*|>

New links

- Enjoy the blog Announcing the Winners
- Read the winning entries on GitHub

Deadlines

- Launch: February 1, 2016
- Submissions close: April 29, 2016
- Announcement of winners: May 16, 2016

The Challenge is now closed.

Official Rules

Read the Official Rules

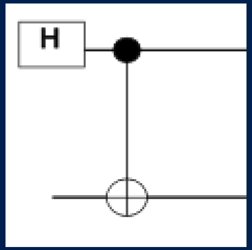
Links

- Register for the Challenge
- Read the FAQ for answers
- Learn about LIQ*Ui*|>
- Watch short videos about LIQ*Ui*|>
- Watch the tutorial video
- Discover the QuArC Group
- Download the simulator

Quantum "Hello World!"

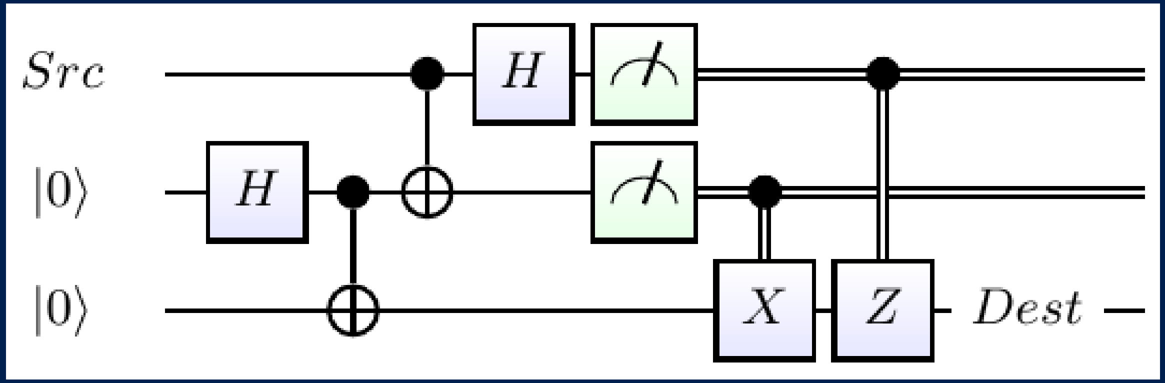
- Define a function to generate entanglement:

```
let EPR (qs:Qubits) = H qs; CNOT qs
```



- The rest of the algorithm:

```
let teleport (qs:Qubits) =
    let qs'      = qs.Tail
    EPR qs'; CNOT qs; H qs
    M qs'; BC X qs'
    M qs ; BC Z !!(qs,0,2)
```



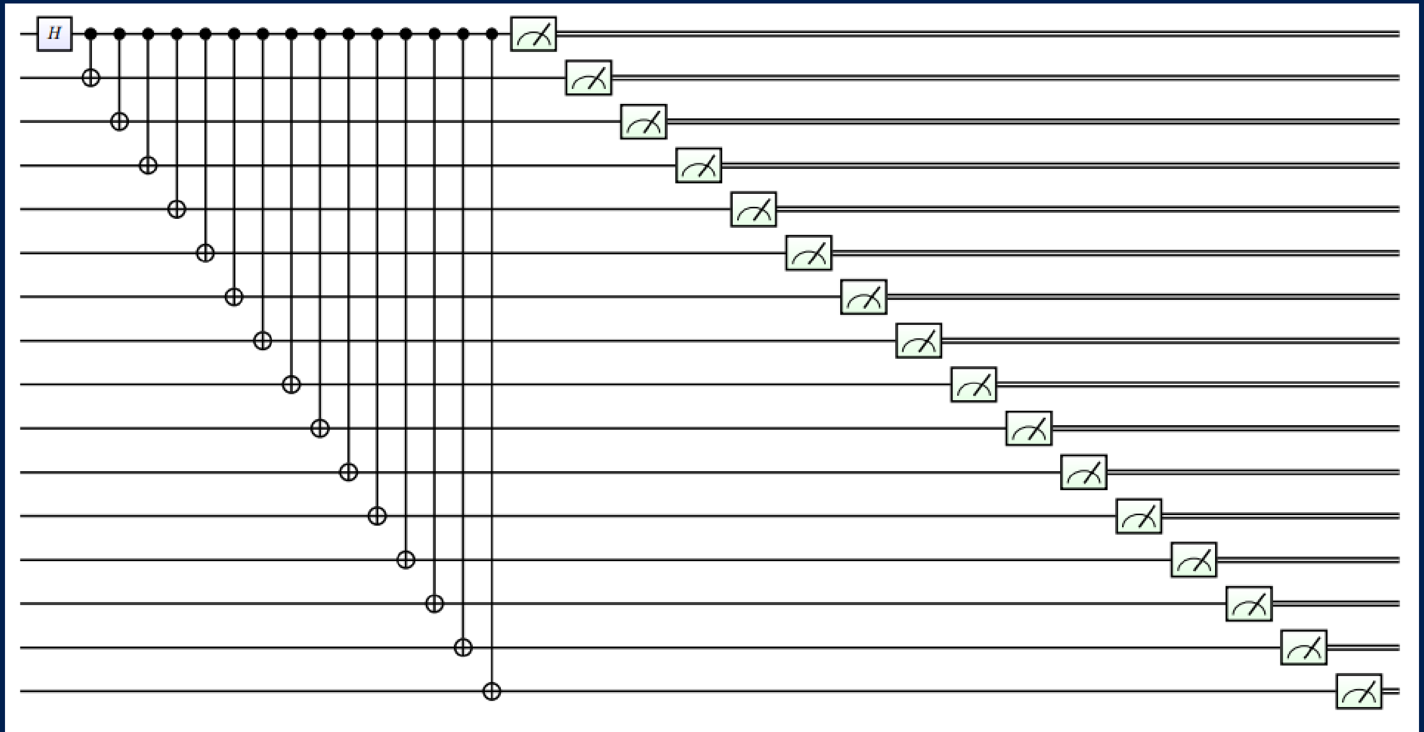
Teleport: running the code

```
loop N times:
        … create 3 qubits
        … init the first one to a random state
        … print it out
        teleport qs
        … print out the result
```

```
0:0000.0/Initial State: (  0.3735-0.2531i)|0>+( -0.4615-0.7639i)|1>
0:0000.0/Final   State: (  0.3735-0.2531i)|0>+( -0.4615-0.7639i)|1> (bits:10)
0:0000.0/Initial State: ( -0.1105+0.3395i)|0>+(   0.927-0.1146i)|1>
0:0000.0/Final   State: ( -0.1105+0.3395i)|0>+(   0.927-0.1146i)|1> (bits:11)
0:0000.0/Initial State: ( -0.3882-0.2646i)|0>+( -0.8092+0.3528i)|1>
0:0000.0/Final   State: ( -0.3882-0.2646i)|0>+( -0.8092+0.3528i)|1> (bits:01)
0:0000.0/Initial State: (  0.2336+0.4446i)|0>+( -0.8527+0.1435i)|1>
0:0000.0/Final   State: (  0.2336+0.4446i)|0>+( -0.8527+0.1435i)|1> (bits:10)
0:0000.0/Initial State: (  0.9698+0.2302i)|0>+(-0.03692+0.0717i)|1>
0:0000.0/Final   State: (  0.9698+0.2302i)|0>+(-0.03692+0.0717i)|1> (bits:11)
0:0000.0/Initial State: (  -0.334-0.3354i)|0>+(   0.315-0.8226i)|1>
0:0000.0/Final   State: (  -0.334-0.3354i)|0>+(   0.315-0.8226i)|1> (bits:01)
```

More complex circuits

```
let entangle (qs:Qubits) =
    H qs; let q0  = qs.Head
    for q in qs.Tail do CNOT[q0;q]
    M >< qs
```

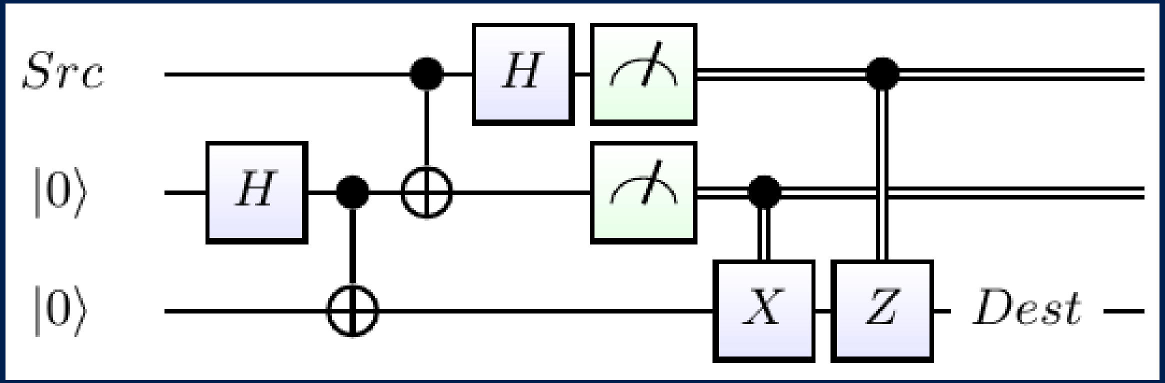


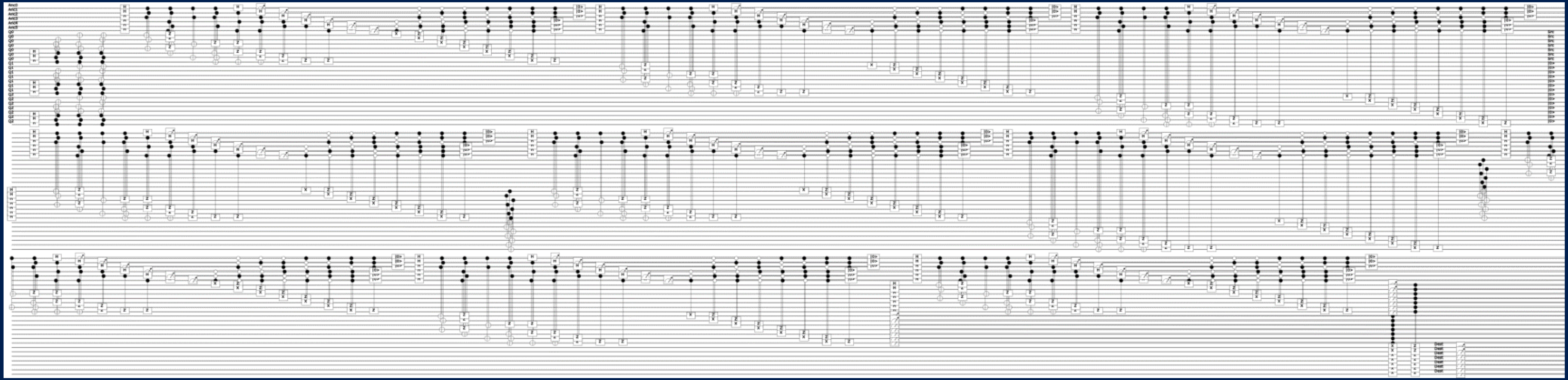

```
0:0000.0/#### Iter   0 [  0.2030]: 0000000000000
0:0000.0/#### Iter   1 [  0.1186]: 0000000000000
0:0000.0/#### Iter   2 [  0.0895]: 0000000000000
0:0000.0/#### Iter   3 [  0.0749]: 0000000000000
0:0000.0/#### Iter   4 [  0.0664]: 1111111111111
0:0000.0/#### Iter   5 [  0.0597]: 0000000000000
0:0000.0/#### Iter   6 [  0.0550]: 1111111111111
0:0000.0/#### Iter   7 [  0.0512]: 0000000000000
0:0000.0/#### Iter   8 [  0.0484]: 0000000000000
0:0000.0/#### Iter   9 [  0.0463]: 0000000000000
0:0000.0/#### Iter  10 [  0.0446]: 0000000000000
0:0000.0/#### Iter  11 [  0.0432]: 1111111111111
0:0000.0/#### Iter  12 [  0.0420]: 0000000000000
0:0000.0/#### Iter  13 [  0.0410]: 0000000000000
0:0000.0/#### Iter  14 [  0.0402]: 0000000000000
0:0000.0/#### Iter  15 [  0.0399]: 0000000000000
0:0000.0/#### Iter  16 [  0.0392]: 1111111111111
0:0000.0/#### Iter  17 [  0.0387]: 1111111111111
0:0000.0/#### Iter  18 [  0.0380]: 0000000000000
0:0000.0/#### Iter  19 [  0.0374]: 1111111111111
```

User defined gates

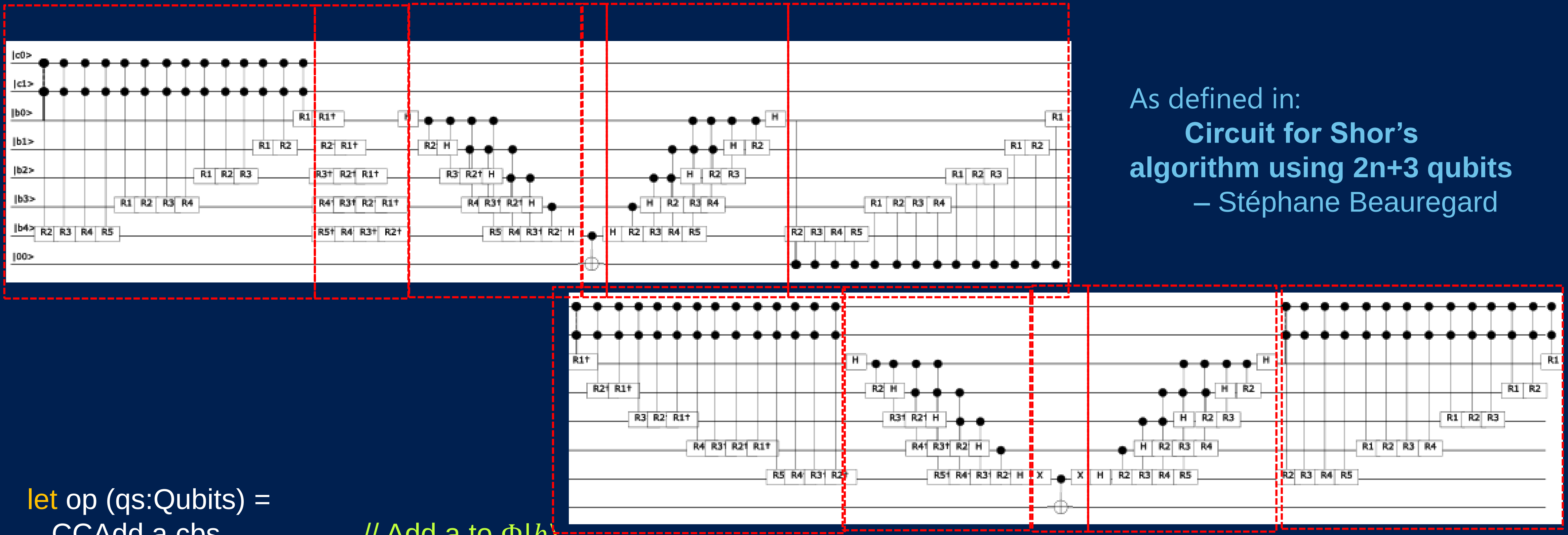
```
/// <summary>
/// Controlled NOT gate
/// </summary>
/// <param name="qs"> Use first two qubits for gate</param>
[<LQD>]
let CNOT (qs:Qubits) =
    let gate =
        Gate.Build("CNOT",fun () ->
            new Gate(
                Name     = "CNOT",
                Help     = "Controlled NOT",
                Mat      = CSMat(4,[(0,0,1.,0.);(1,1,1.,0.);
                                    (2,3,1.,0.);(3,2,1.,0.)]),
                Draw     = "\\ctrl{#1}\\go[#1]\\targ"
        ))
    gate.Run qs
```

Full teleport circuit in a Steane7 code





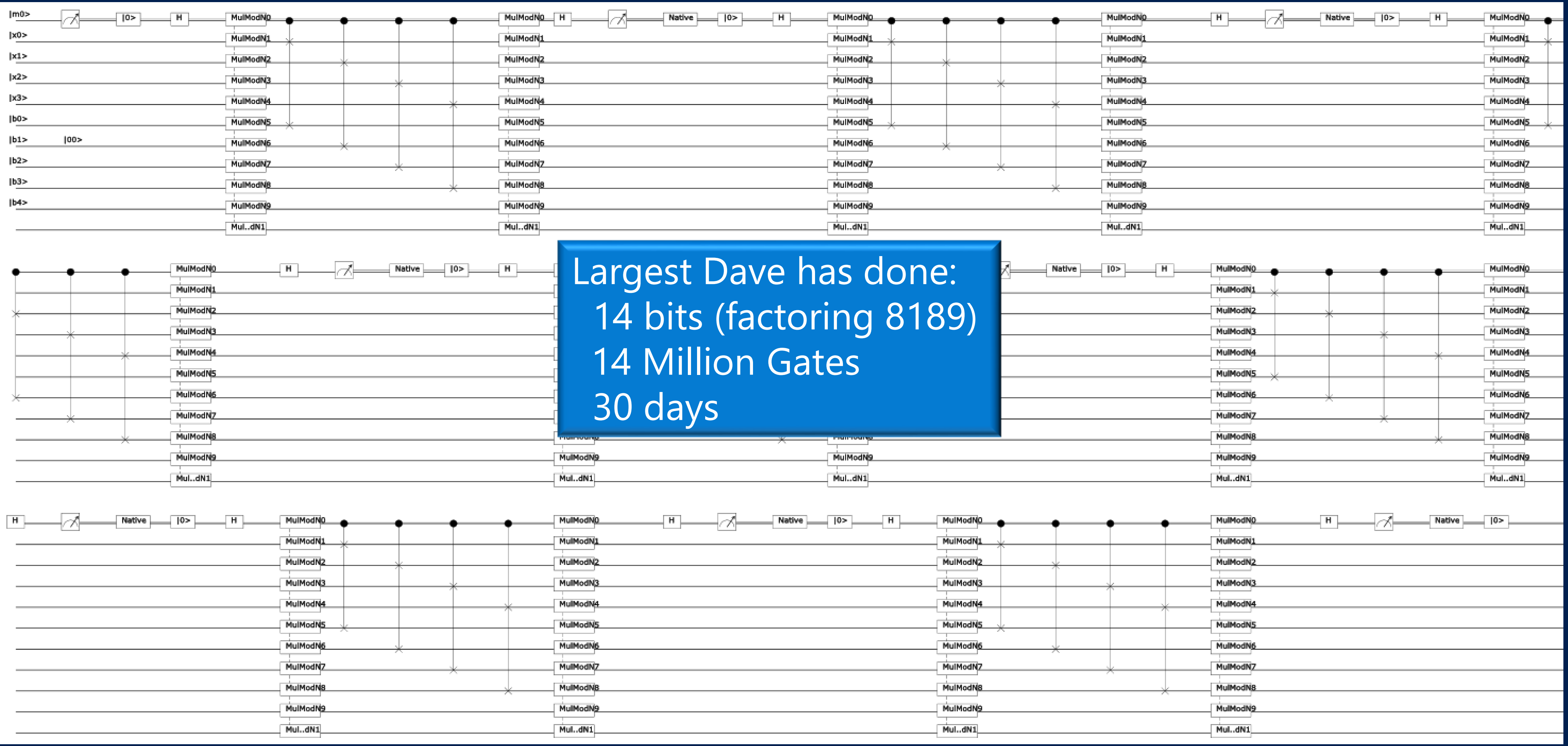
Shor's algorithm component: modular adder



As defined in:
Circuit for Shor's algorithm using 2n+3 qubits
– Stéphane Beauregard

```
let op (qs:Qubits) =
    CCAdd a cbs          // Add a to Φ|b⟩
    AddA' N bs           // Sub N from Φ|a + b⟩
    QFT' bs              // Inverse QFT of Φ|a + b − N⟩
    CNOT [bMx;anc]       // Save top bit in Ancilla
    QFT bs               // QFT of a+b-N
    CAddA N (anc :: bs)  // Add back N if negative
    CCAdd' a cbs         // Subtract a from Φ|a + b mod N⟩
    QFT' bs              // Inverse QFT
    X [bMx]              // Flip top bit
    CNOT [bMx;anc]       // Reset Ancilla to |0⟩
    X [bMx]              // Flip top bit back
    QFT bs               // QFT back
    CCAdd a cbs          // Finally get Φ|a + b mod N⟩
```

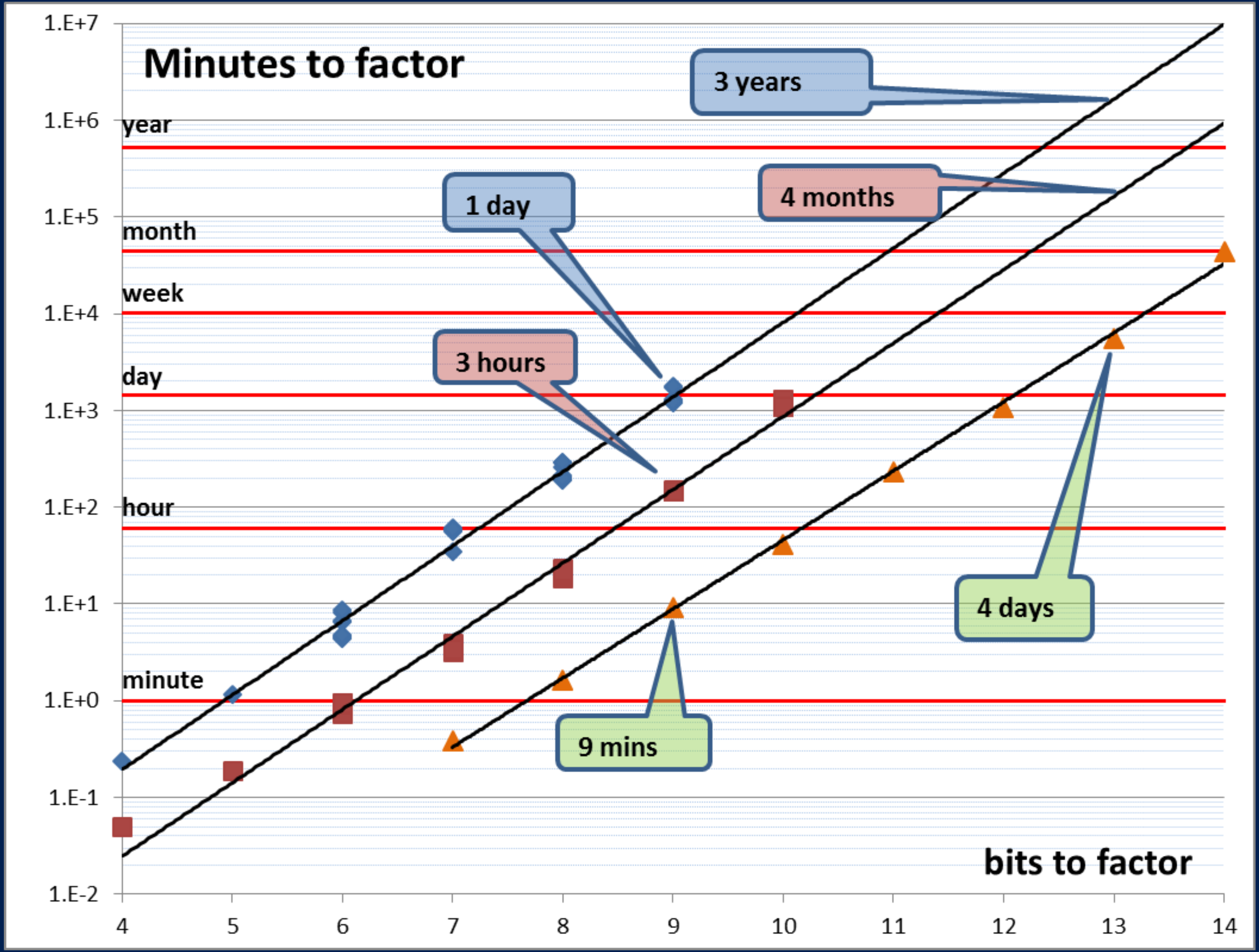
Shor's algorithm: full circuit: 4 bits ≅ 8200 gates



Largest Dave has done:
- 14 bits (factoring 8189)
- 14 Million Gates
- 30 days

Circuit for Shor's algorithm using 2n+3 qubits – Stéphane Beauregard

Shor's algorithm: scaling



LIQ$Ui|\rangle$ - Optimizations

- If we can guarantee that the qubits we want to operate on are always at the beginning of the state vector, we can view the operation as:

$$G_{2^k,2^k} \otimes I_{2^{n-k},2^{n-k}} \times \Psi_{2^n}$$

- However, what we'd really like is to flip the Kronecker product order:

$$I_{2^{n-k},2^{n-k}} \otimes G_{2^k,2^k} \times \Psi_{2^n}$$

- This would accomplish :
  - $I \otimes G$ would become a block diagonal matrix that just has copies of G down the diagonal. This means that you'd never have to actually materialize $U = I \otimes G$
  - Processing would be highly parallel (and/or distributed) because the matrix is perfectly partitioned and applies to separate, independent parts of the state vector

Quantum Chemistry

$$H = \sum_{pq} h_{pq} a_p^\dagger a_q + \frac{1}{2} \sum_{pqrs} h_{pqrs}\, a_p^\dagger a_q^\dagger a_r a_s$$

Can quantum chemistry be performed on a small quantum computer: D
Hastings, Ma

Impro
Has

As quantum
computers w
appear feasil
applications
frequently m
simulating q
of molecules
computation
perform qua
the quantum
molecule twi
exactly. We f
increase in th
required incr
executed is r
quantum cor
problems, dr
http://arxiv.c

W
al
co
in
nu
ac
ca
de
pe
nu
Tr
Tr
to
of
de
h

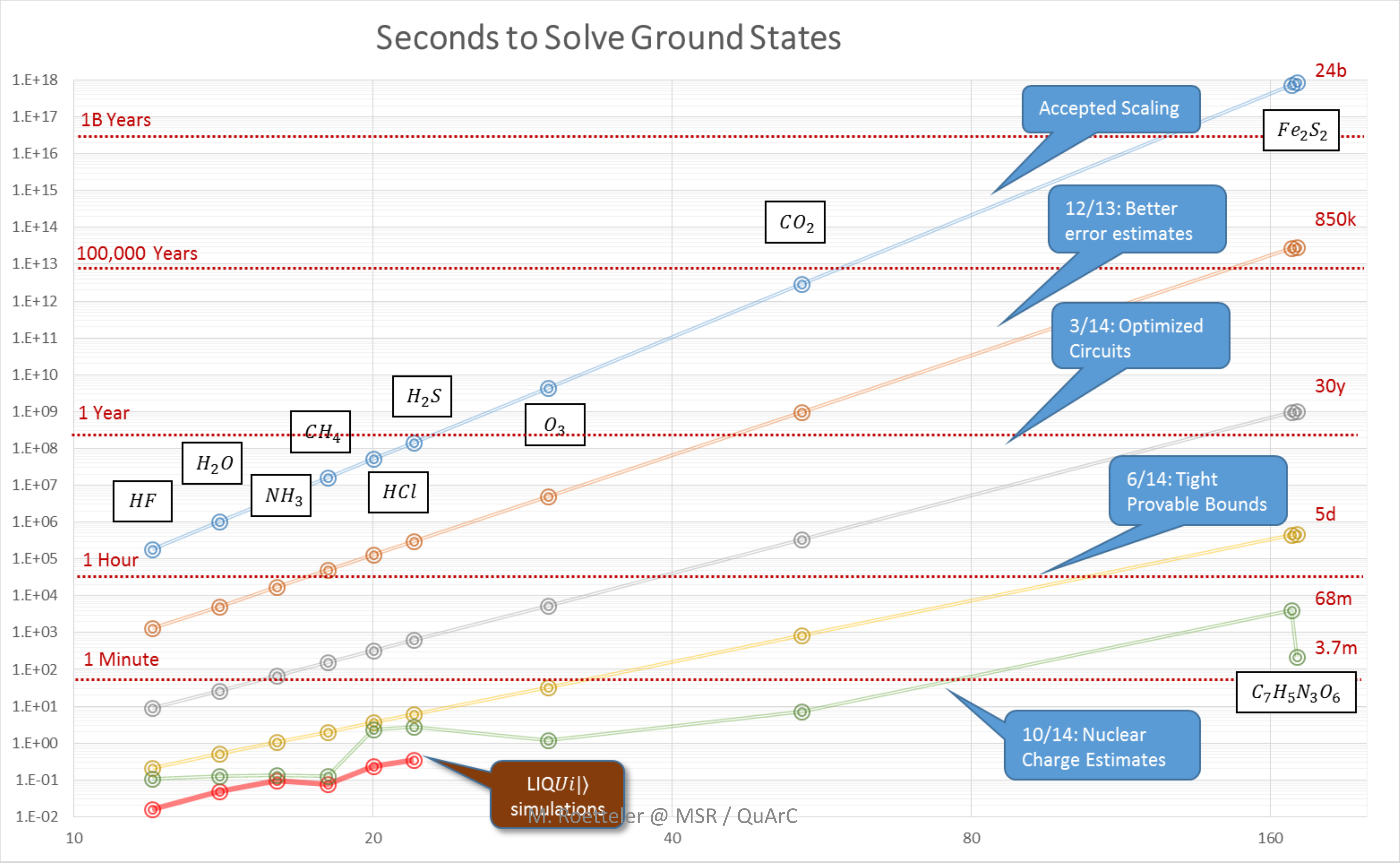
ions of
lgorithms
of the
ever, we
me
relation
ical
of orbitals,
tes several
nd to
ces. Finally,
totically

Ferredoxin (Fe_2S_2) used in many metabolic reactions including energy transport in photosynthesis

- *Intractable on a classical computer*
- *Assumed quantum scaling: ~24 billion years (N^{11} scaling)*
- *First paper: ~850 thousand years to solve (N^9 scaling)*
- *Second paper: ~30 years to solve (N^7 scaling)*
- *Third paper: ~5 days to solve ($N^{5.5}$ scaling)*
- *Fourth paper: ~1 hour to solve ($N^3, Z^{2.5}$ scaling)*

Quantum Chemistry

$$H = \sum_{pq} h_{pq} a_p^\dagger a_q + \frac{1}{2} \sum_{pqrs} h_{pqrs}\, a_p^\dagger a_q^\dagger a_r a_s$$



Quantum and reversible circuit synthesis

Instruction sets: universal single-qubit bases

- T + Clifford (H, X, Y, Z, I, S)

$$T = R\left(\frac{\pi}{4}\right) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$$

- V_3 + Clifford (H, X, Y, Z, I, S)

$$V_3 = \frac{1}{\sqrt{5}} \begin{bmatrix} 1+2i & 0 \\ 0 & 1-2i \end{bmatrix}$$

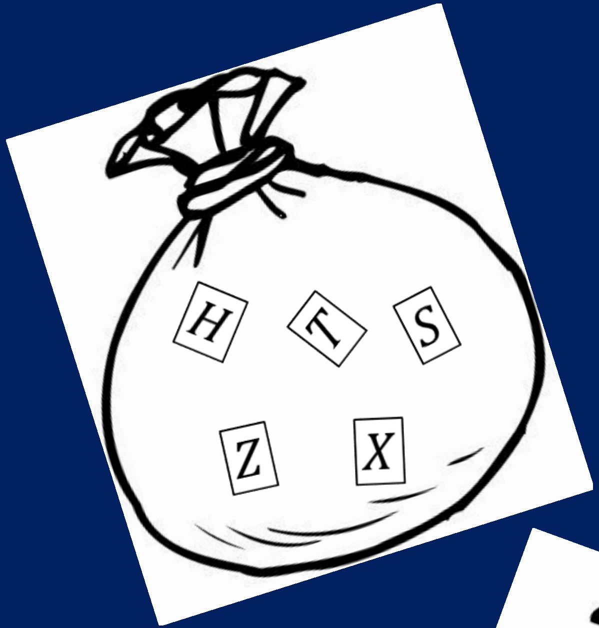
- $\frac{\pi}{12}$ + Clifford (H, X, Y, Z, I, S)

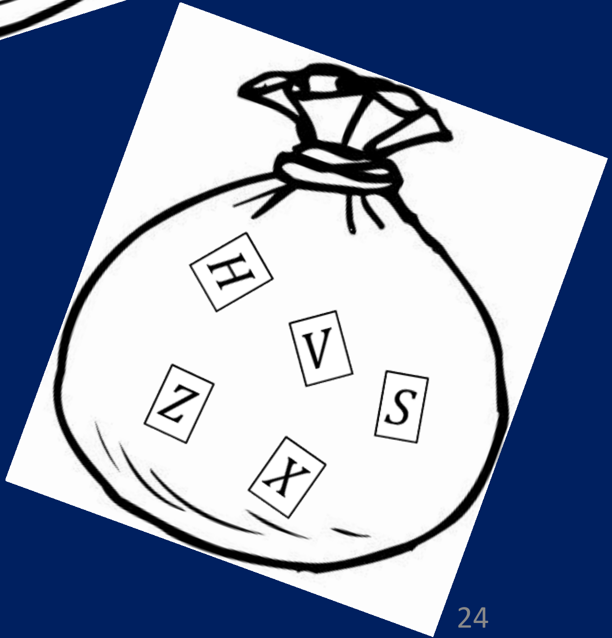
$$R\left(\frac{\pi}{6}\right) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/6} \end{bmatrix}$$

- Fibonacci anyon basis:

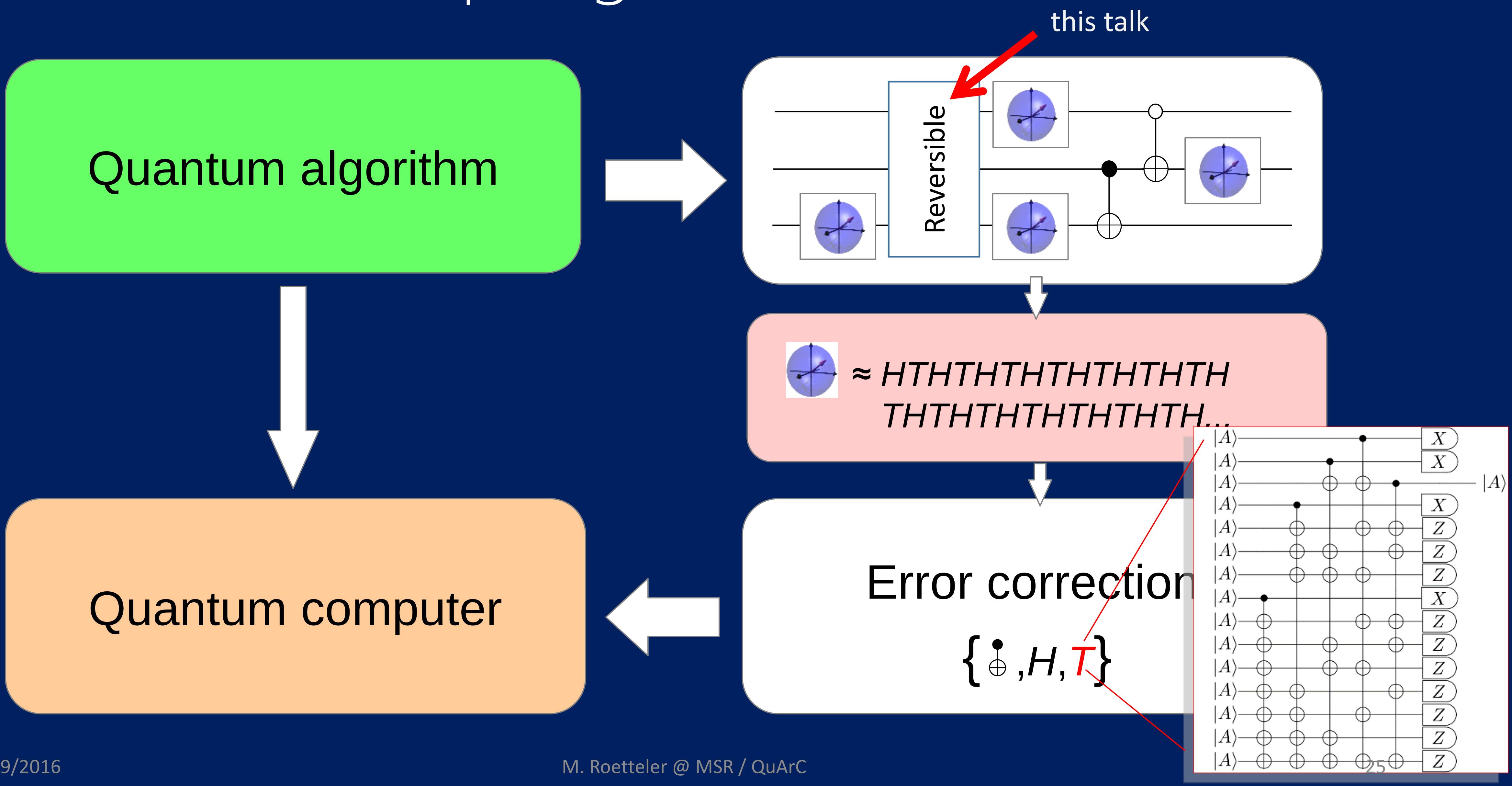
$$\sigma_1 = \begin{bmatrix} -\omega & 0 \\ 0 & \omega^3 \end{bmatrix}, \sigma_2 = \begin{bmatrix} \omega^4\tau & -\omega^2\sqrt{\tau} \\ -\omega^2\sqrt{\tau} & -\tau \end{bmatrix},$$

$$\omega = e^{i\pi/5}, \tau = \frac{\sqrt{5}-1}{2}$$

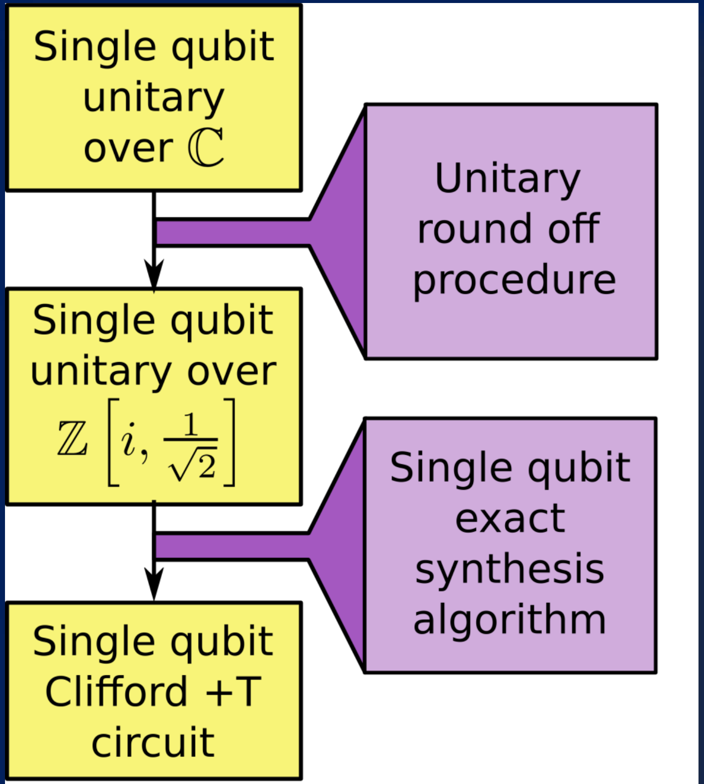




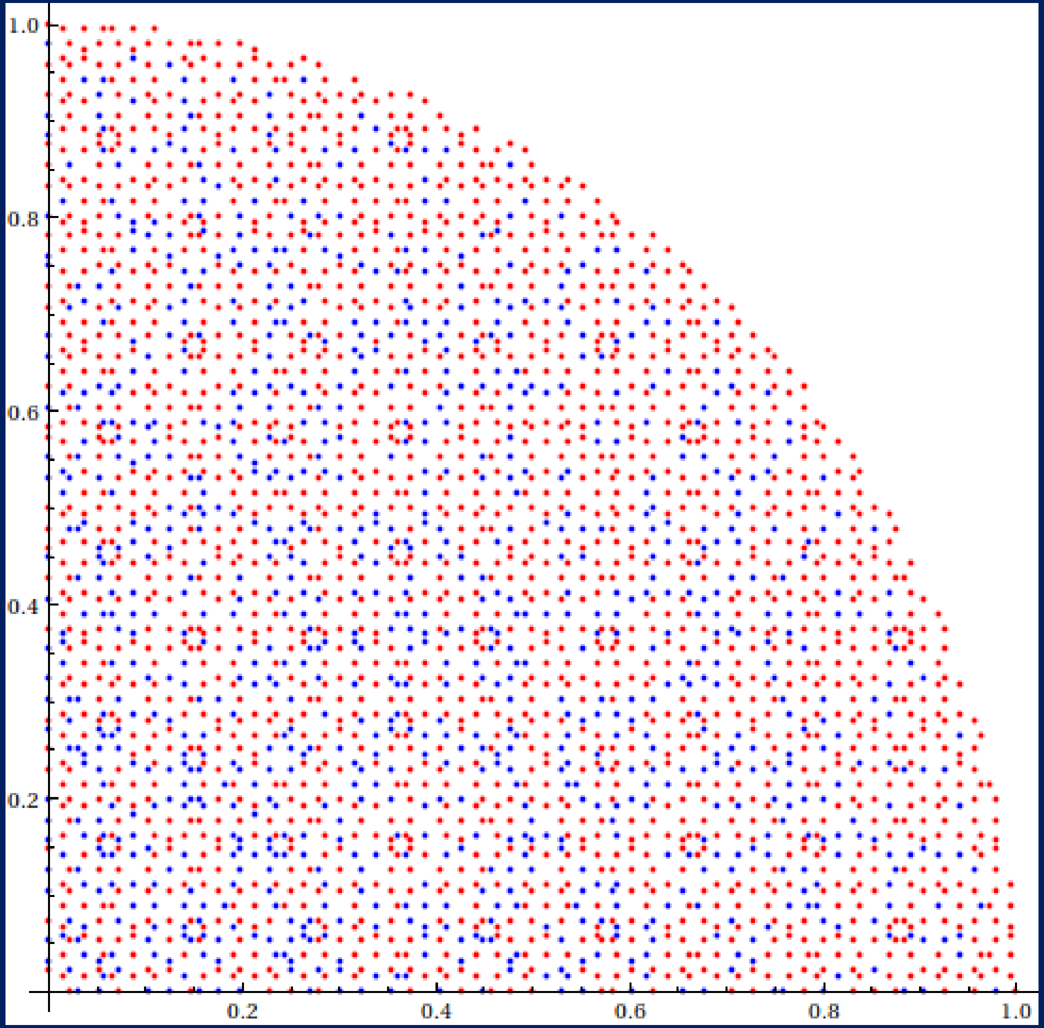
Quantum compiling



Year 2012: Revolution in synthesis methods! (based on algebraic number theory)



Number of T gates required is $O\left(\log\left(1/\varepsilon\right)\right)$ vs $O\left(\log^{3+\delta}\left(1/\varepsilon\right)\right)$ (for the Solovay-Kitaev algorithm)



[Kliuchnikov/Maslov/Mosca'12], [Selinger'12], [Ross/Selinger'14], [Kliuchnikov/Yard'15]

Reversible computing: why bother?

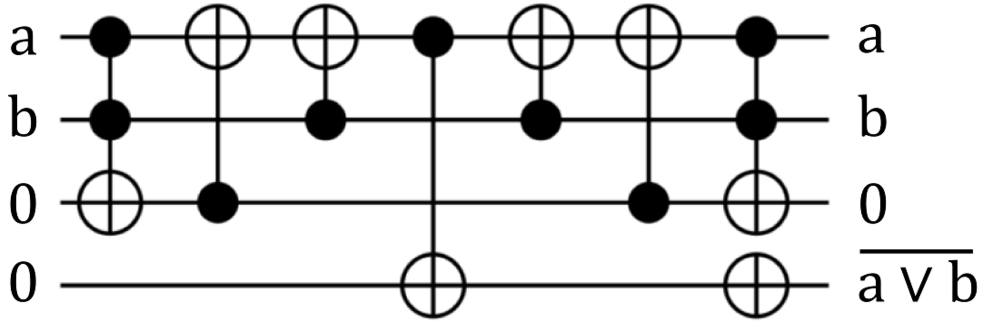
- Arithmetic:
  - Factoring: just needs "constant" modular arithmetic
  - ECC dlogs: need generic modular arithmetic
  - HHL: need integer inverses; Newton type methods
- Amplitude amplification:
  - Implementation of the "oracles", e.g., for search, collision etc.
  - Implementation of walk operators on data structures
- Quantum simulation:
  - Addressing/indexing functions for sparse matrices
  - Computing Hamiltonian terms on the fly

See also: "lifting monad" in Quipper

Universal gate set: Toffoli gates

Fact: The set {Toffoli, CNOT, NOT} is universal for reversible computing: any *even* permutation on n qubits can be written as a sequence of Toffoli, CNOT, and NOT gates. **[Toffoli'80], [Fredkin/Toffoli'82]**

Example:



Main motivation: How can we find efficient implementations of reversible circuits in terms of efficient Toffoli networks?
How can we do this starting from irreversible descriptions in a programming language like Python or Haskell or F# or C?
Can we trade time (circuit depth) for space (#qubits) in a meaningful way?

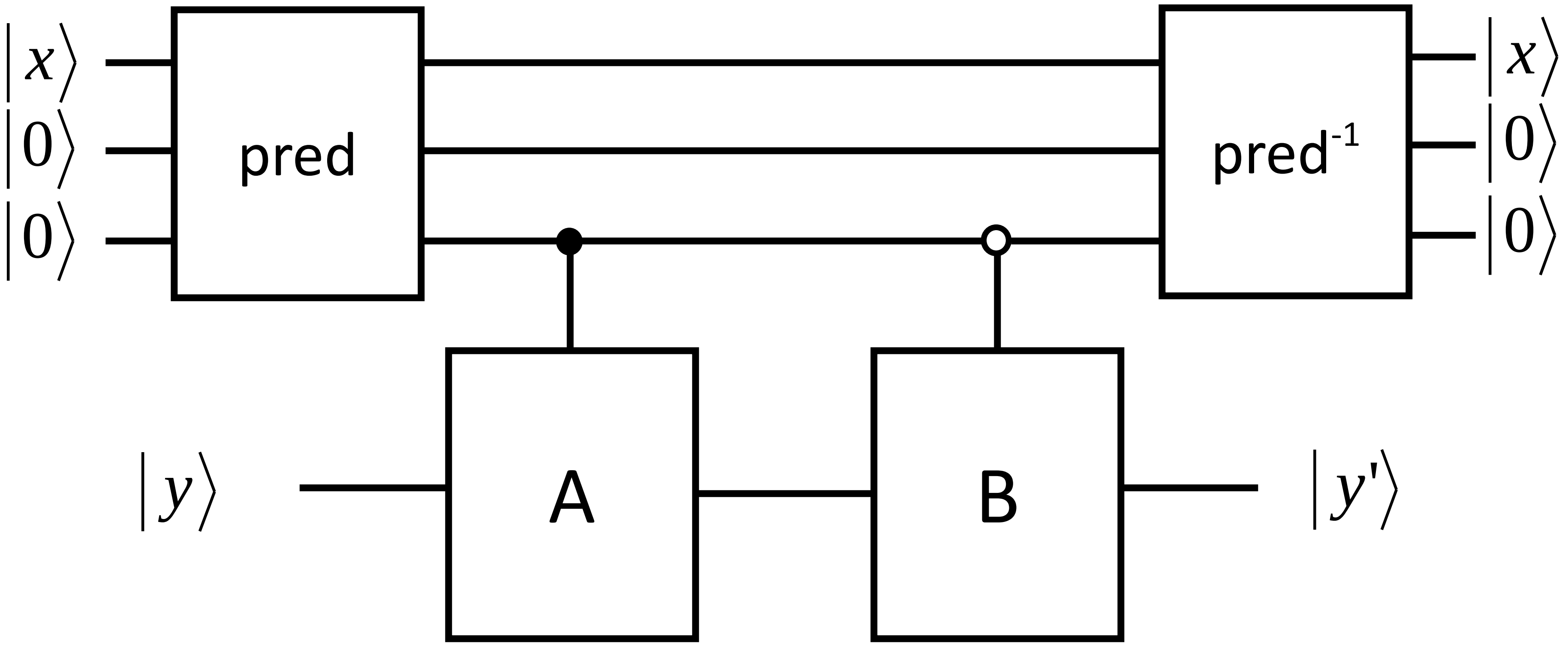
Example: Carry ripple adder (in F#)

```
let carryRippleAdder (a:bool []) (b:bool []) =
   let n = Array.length a
   let result =  Array.zeroCreate (n)
   result.[0] <- a.[0] <> b.[0]
   let mutable carry = a.[0] && b.[0]
   result.[1] <- a.[1] <> b.[1] <> carry
   for i in 2 .. n - 1 do
      // compute outgoing carry from current bits and incoming carry
      carry <- (a.[i-1] && (carry <> b.[i-1])) <> (carry && b.[i-1])
      result.[i]  <-  a.[i] <> b.[i] <> carry
   result
```

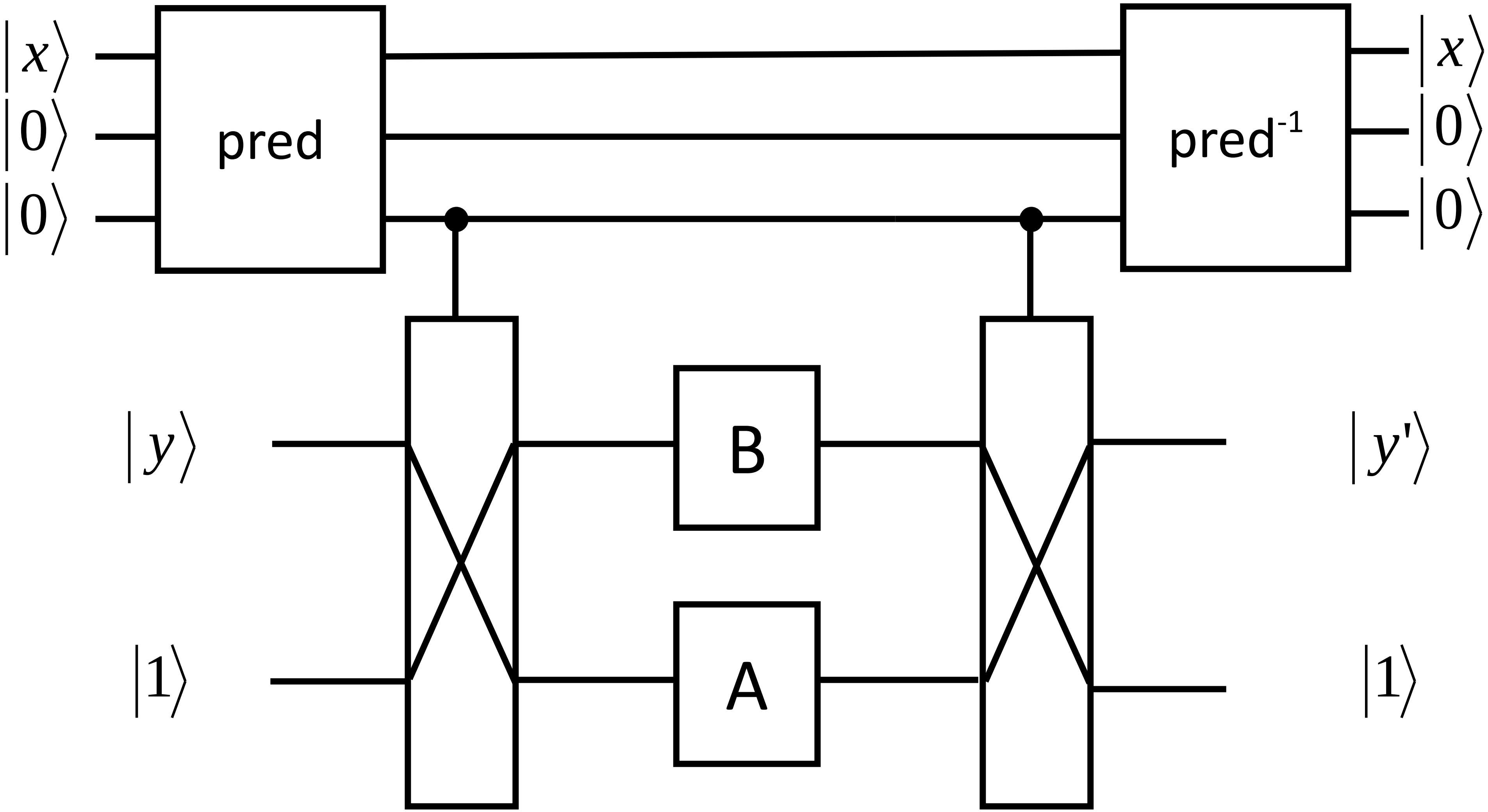
Example: If-then-else expressions

```
// module emission_tst_workaround: float -> float -> unit
// author = MG_Burns, changeset = 1519992, date = 06/03/2009

let THRTTL_MIN = 1.0
let THRTTL_MAX = 49.9

let emission_tst_workaround (v_front_wheels:float) (v_rear_wheels:float) =
    let epa_detect = (v_front_wheels > 0.0) && (v_rear_wheels = 0.0)
    if epa_detect then
        let throttleSettings = THRTTL_MIN
        let catConverterOn = true
    else                                   // MGB: just like taking candy from a baby
        let throttleSettings = THRTTL_MAX
        let catConverterOn = false
    runEngine throttleSettings catConverterOn
```

If-then-else construct I

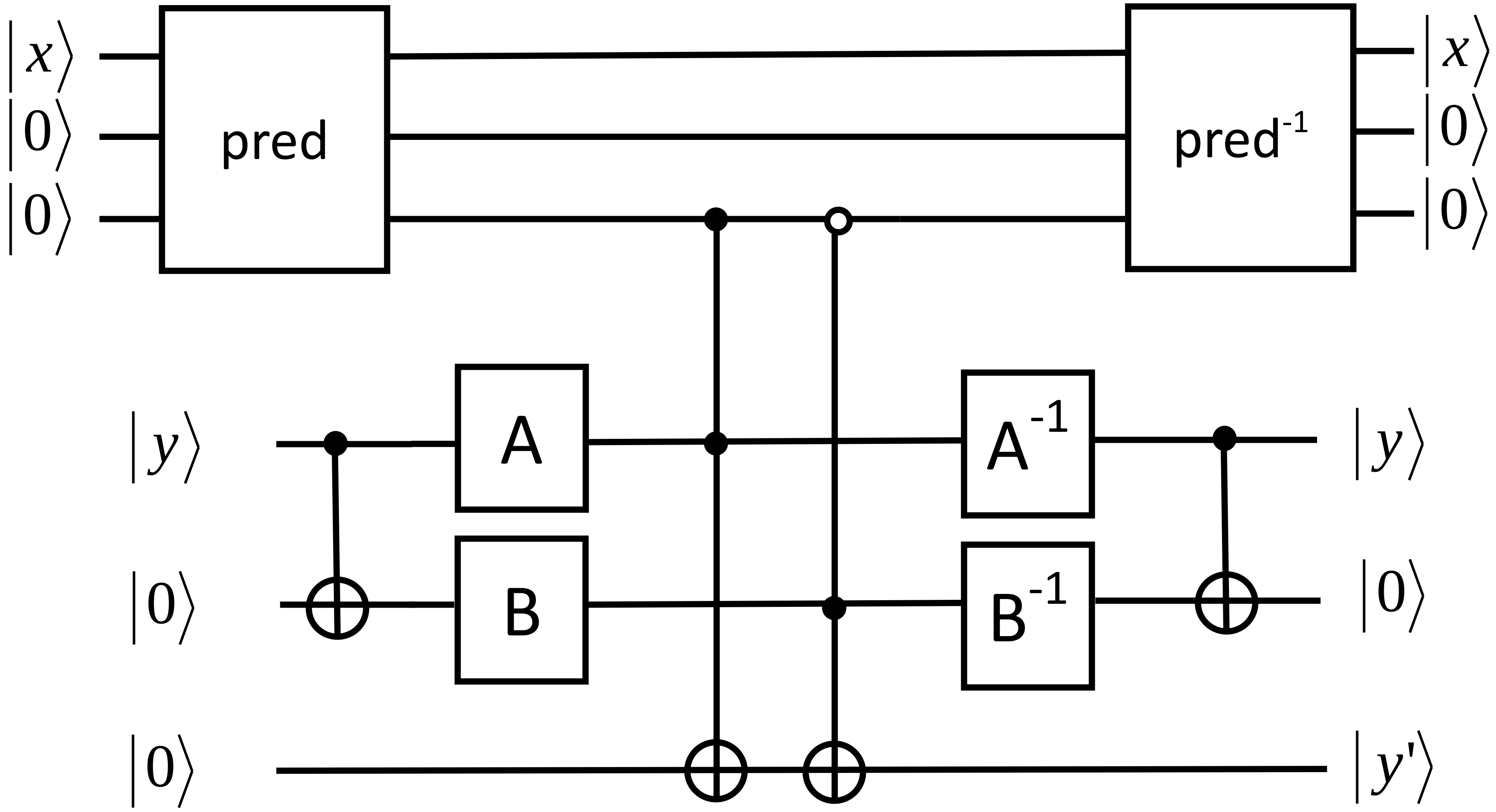


If-then-else construct II



[Maslov, Saeedi ‘01]

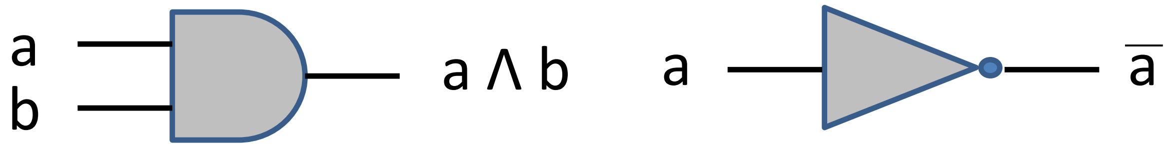
If-then-else construct III



Reversible computing: at the gate level

- We assume that function is given as **combinational circuits**, i.e., circuits that do not make use of memory elements or feedback.

- Universal families of irreversible gates:

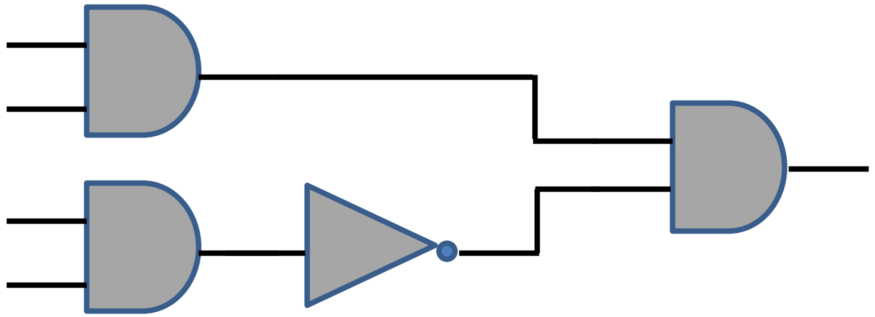


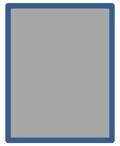
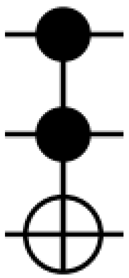
- We can compose gates together to make larger circuits.

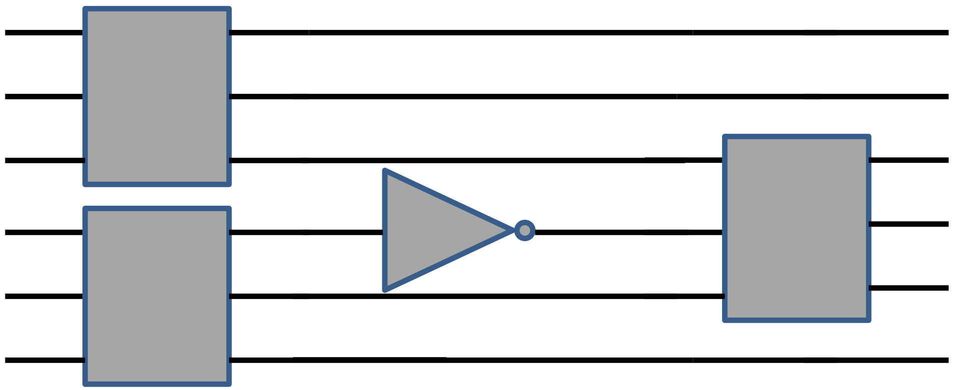
- Basic issue: many gates are <u>not</u> reversible!

Reversible computing: at the gate level

Example:

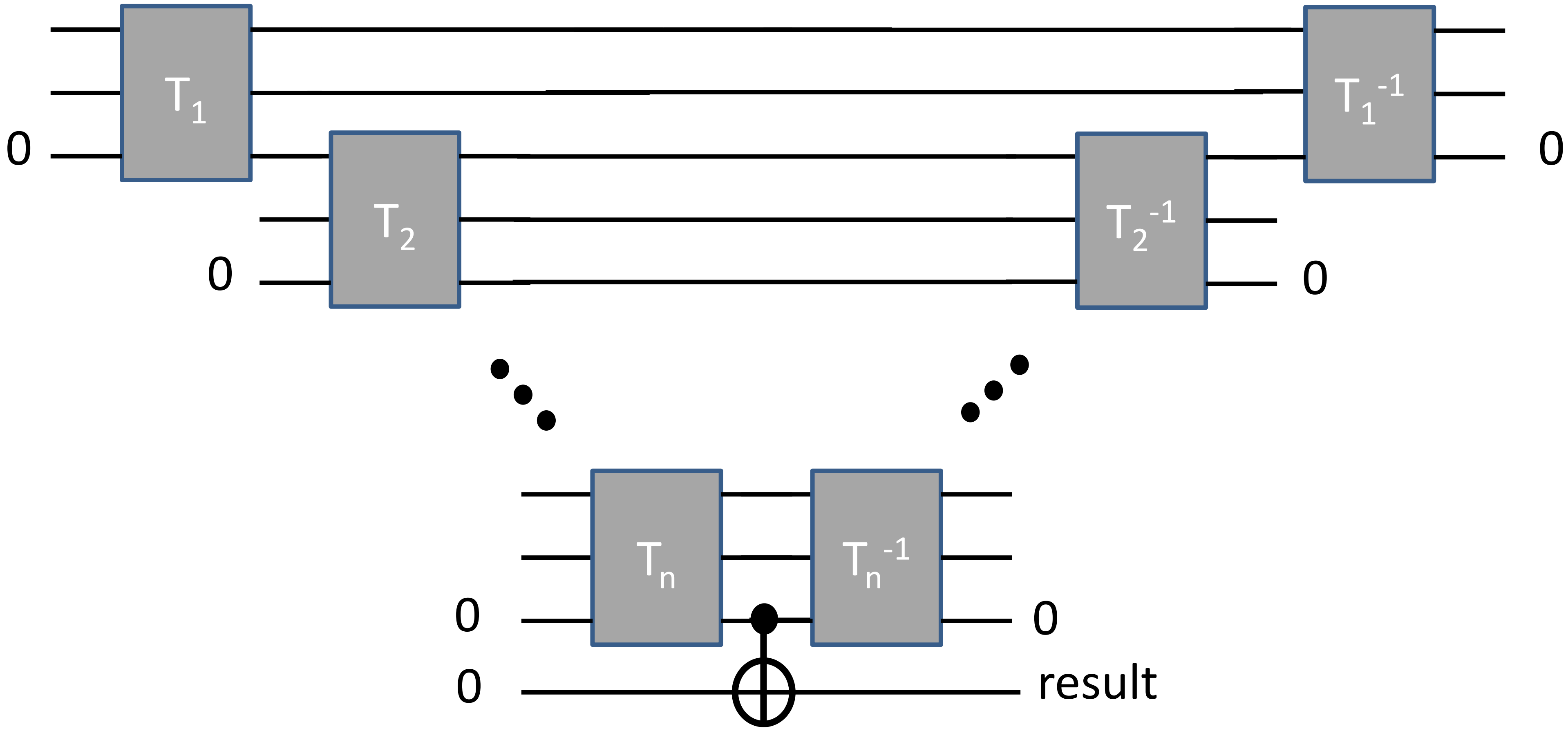


Replace each gate with a reversible one: (e.g.  = Toffoli gate )



Cleaning up the scratch bits

Replace each gate with a reversible one **[Bennett, IBM JRD'73]:**

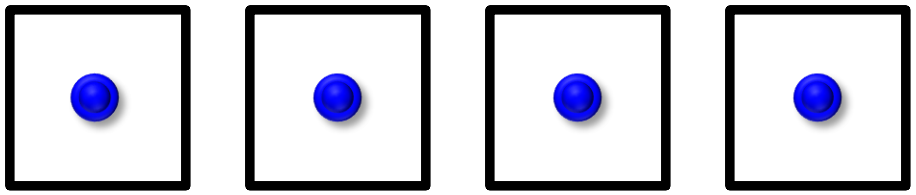


Pebble game: case of 1D graph

Rules of the game: **[Bennett, SIAM J. Comp., 1989]**

- n boxes, labeled i = 1, …, n
- in each move, either add or remove a pebble
- a pebble can be added or removed in i=1 at any time
- a pebble can be added of removed in i>1 if and only if there is a pebble in i-1
- 1D nature arises from decomposing a computation into “stages”

Example:



1 2 3 4

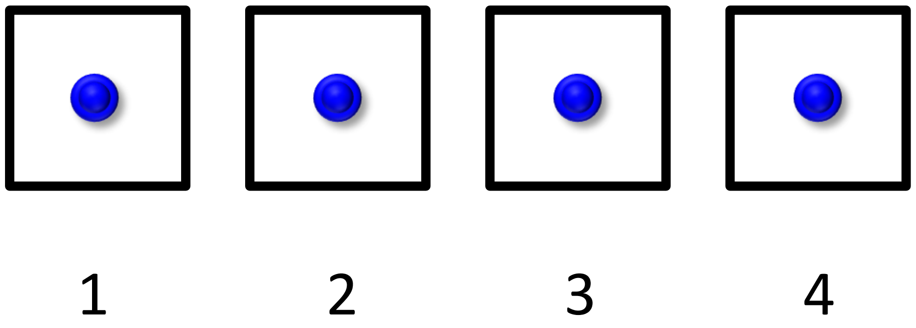
#	i
1	1
2	2
3	3
4	4
5	3
6	2
7	1

Pebble game: 1D plus space constraints

Imposing resource constraints:

- only a total of S pebbles are allowed
- corresponds to reversible algorithm with at most S ancilla qubits

Example: (n=3, S=3)



#	i
1	1
2	2
3	3
4	1
5	4
6	3
7	1
8	2
9	1

Optimal pebbling strategies

Definition: Let X be solution of pebble game. Let T(X) be # steps and Let S(X) be #pebbles. Define F(n,S) = min { T(X) : S(X) ≤ S }.

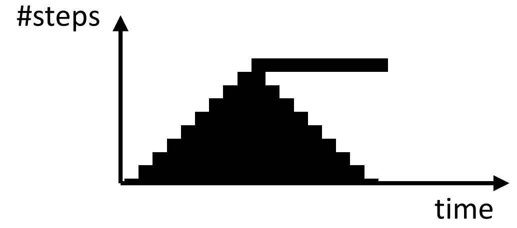
Table (small values of F):

$n \setminus S$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	∞	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
3	∞	∞	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
4	∞	∞	9	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
5	∞	∞	∞	11	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
6	∞	∞	∞	15	13	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
7	∞	∞	∞	19	17	15	13	13	13	13	13	13	13	13	13	13	13	13	13	13
8	∞	∞	∞	25	21	19	17	15	15	15	15	15	15	15	15	15	15	15	15	15
9	∞	∞	∞	∞	25	23	21	19	17	17	17	17	17	17	17	17	17	17	17	17
10	∞	∞	∞	∞	29	27	25	23	21	19	19	19	19	19	19	19	19	19	19	19
11	∞	∞	∞	∞	33	31	29	27	25	23	21	21	21	21	21	21	21	21	21	21
12	∞	∞	∞	∞	39	35	33	31	29	27	25	23	23	23	23	23	23	23	23	23
13	∞	∞	∞	∞	45	39	37	35	33	31	29	27	25	25	25	25	25	25	25	25
14	∞	∞	∞	∞	53	43	41	39	37	35	33	31	29	27	27	27	27	27	27	27
15	∞	∞	∞	∞	61	47	45	43	41	39	37	35	33	31	29	29	29	29	29	29
16	∞	∞	∞	∞	71	51	49	47	45	43	41	39	37	35	33	31	31	31	31	31
17	∞	∞	∞	∞	∞	57	53	51	49	47	45	43	41	39	37	35	33	33	33	33
18	∞	∞	∞	∞	∞	63	57	55	53	51	49	47	45	43	41	39	37	35	35	35
19	∞	∞	∞	∞	∞	69	61	59	57	55	53	51	49	47	45	43	41	39	37	37
20	∞	∞	∞	∞	∞	77	65	63	61	59	57	55	53	51	49	47	45	43	41	39
21	∞	∞	∞	∞	∞	85	69	67	65	63	61	59	57	55	53	51	49	47	45	43
22	∞	∞	∞	∞	∞	93	73	71	69	67	65	63	61	59	57	55	53	51	49	47
23	∞	∞	∞	∞	∞	101	79	75	73	71	69	67	65	63	61	59	57	55	53	51
24	∞	∞	∞	∞	∞	109	85	79	77	75	73	71	69	67	65	63	61	59	57	55

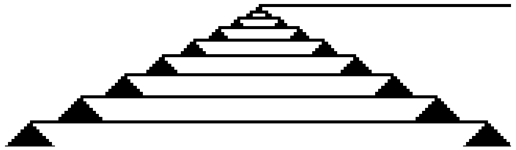
[E.Knill, arxiv:math/9508218]

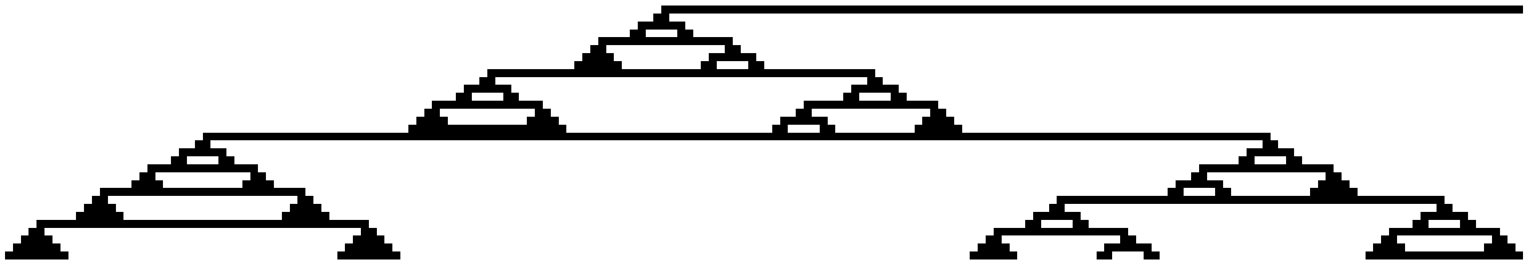
Optimal pebbling strategies: 1D chains

Dynamic programming: Allowed us to find best strategy for given number of steps n to be performed and given space resource constraint S which is the number of available pebbles.

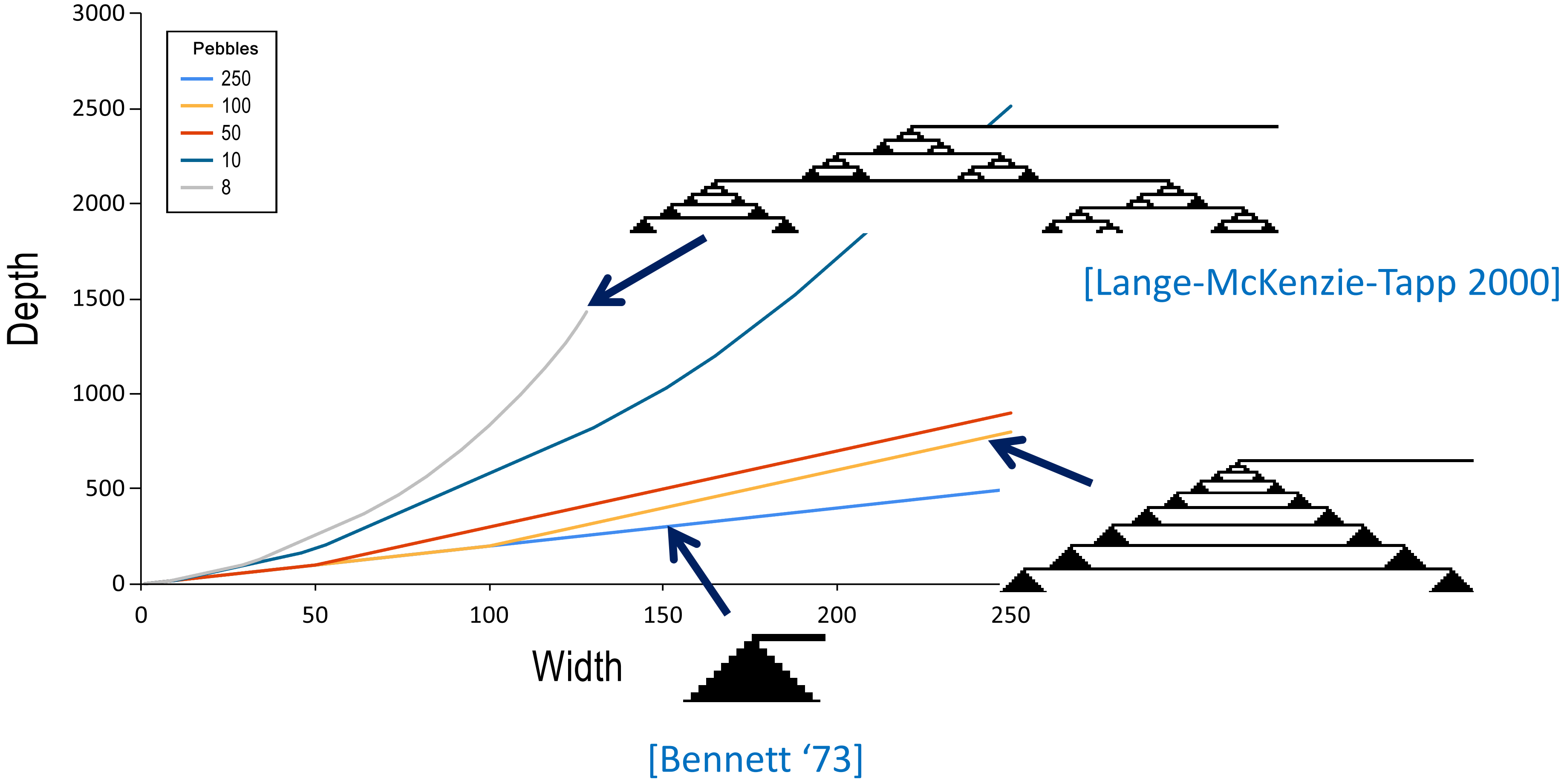


This works ok for 1D chains. For general graphs the problem of finding the optimal strategy is difficult (PSPACE complete problem) -> need heuristics





Optimal pebbling strategies: 1D chains



Time-space tradeoffs

Let A be an algorithm with time complexity T and space complexity S.

- Using reversible pebble game, **[Bennett, SIAM J. Comp. 1989]** showed that for any ε>0 there is a reversible algorithm with time $O(T^{1+\varepsilon})$ and space complexity O(S ln(T)).

- Issue: one cannot simply take the limit ε→0. The space would grow in an unbounded way (as $O(\varepsilon 2^{1/\varepsilon} S \ln(T))$).

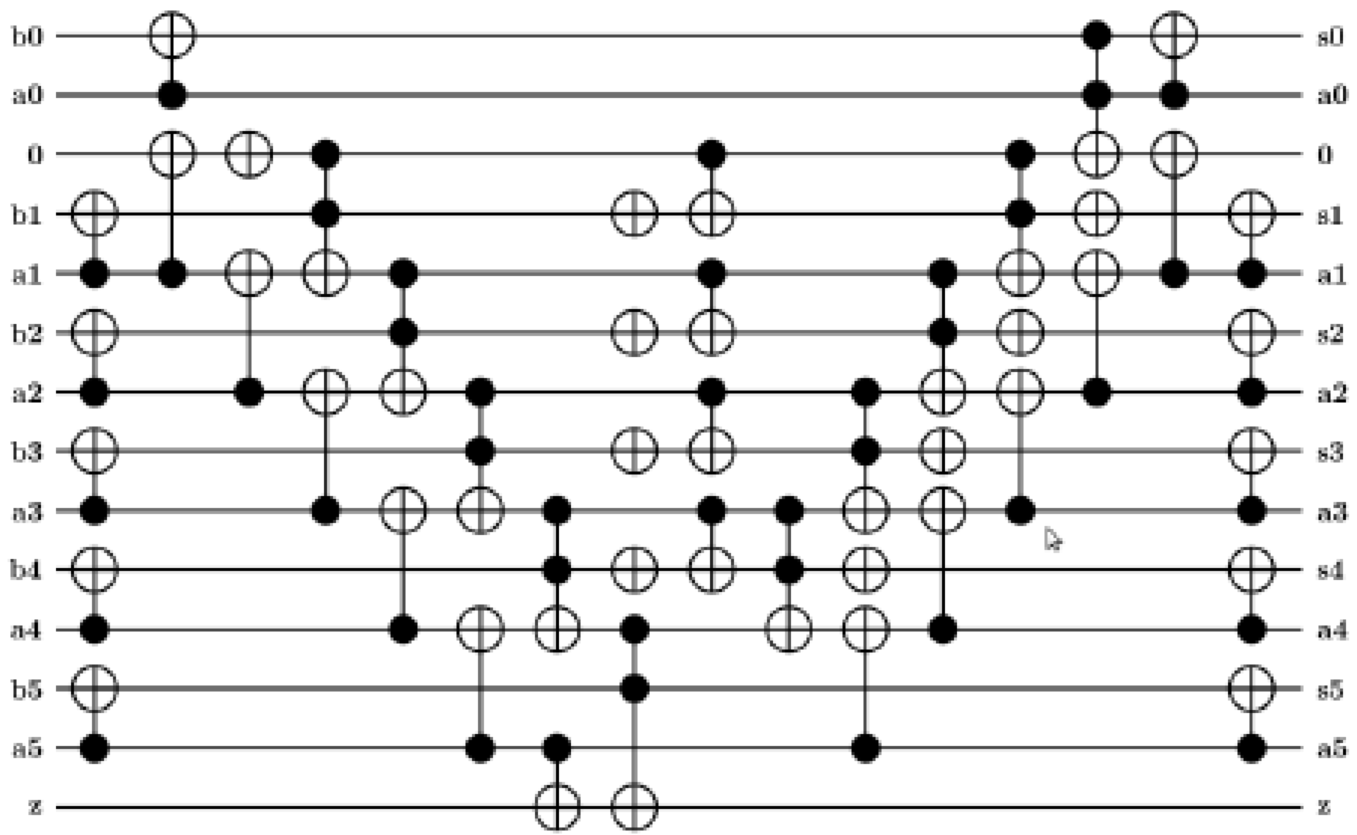
- Improved analysis **[Levine, Sherman, SIAM J. Comp. 1990]** showed that for any ε>0 there is a reversible algorithm time $O(T^{1+\varepsilon}/S^{\varepsilon})$ and space complexity O(S (1+ln(T/S))).

- Other time/space tradeoffs: **[Buhrman, Tromp, Vitányi, ICALP'01]**
  $T_{rev} = S\, 3^k\, 2^{O(T/2^k)}$, $S_{rev} = O(kS)$, where k = #pebbles
  special cases: k = O(1) → **[Lange-McKenzie-Tapp, 2000]**
  k = log T → **[Bennett, 1989]**

- Pebble games played on general DAGs hard to analyze (opt #pebbles = PSPACE complete)
  → need heuristics to tackle general dependency graphs!

New technique:
Mutable data flow analysis

Mutability via in-place operations: e.g. adders

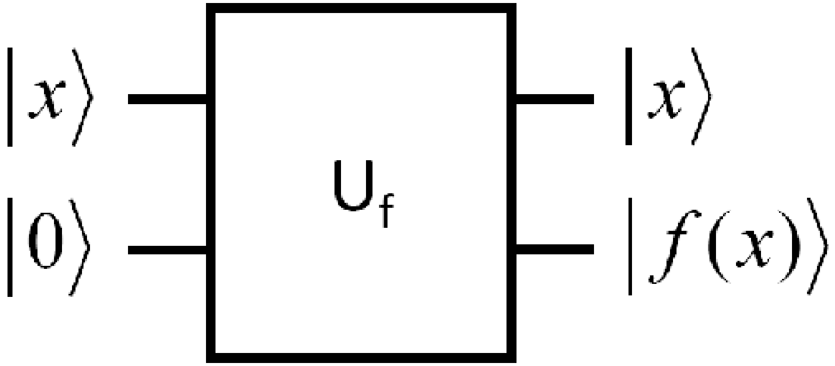


[CDKM:04] S. A. Cuccaro, T. G. Draper, S. A. Kutin, and D. P. Moulton, quant-ph/0410184 (2004).

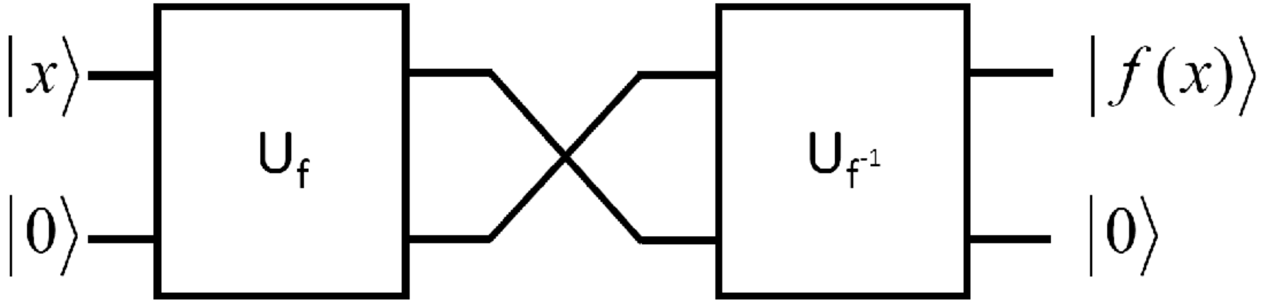
- This is an example for in-place operation (x,y) → (x,x+y)
- At the program level, mutable data can be identified (e.g. via `mutable`)

Manufacturing more in-place computations

Out-of-place circuit for f:



Generic circuit identity: [Kashefi et al], [Mosca et al] describe method that allows in-place efficient computation of f, provided that the inverse has an efficient circuit too.

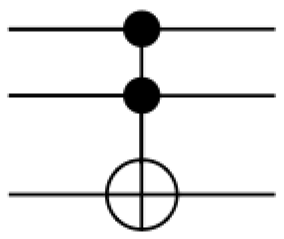


Mutable data dependency graph (MDD)

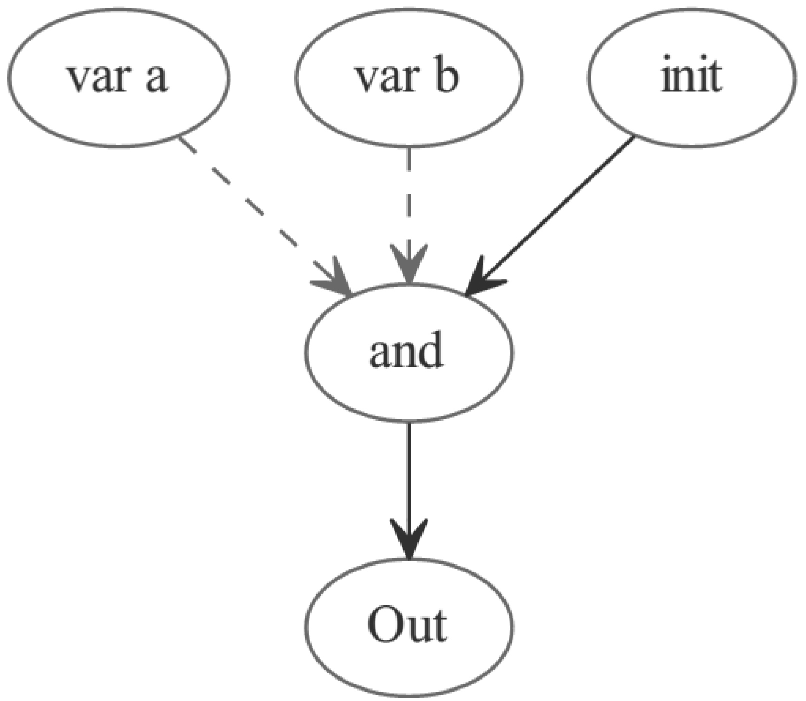
Example:

```
let f a b = a && b
```

Corresponding circuit:



Corresponding MDD:

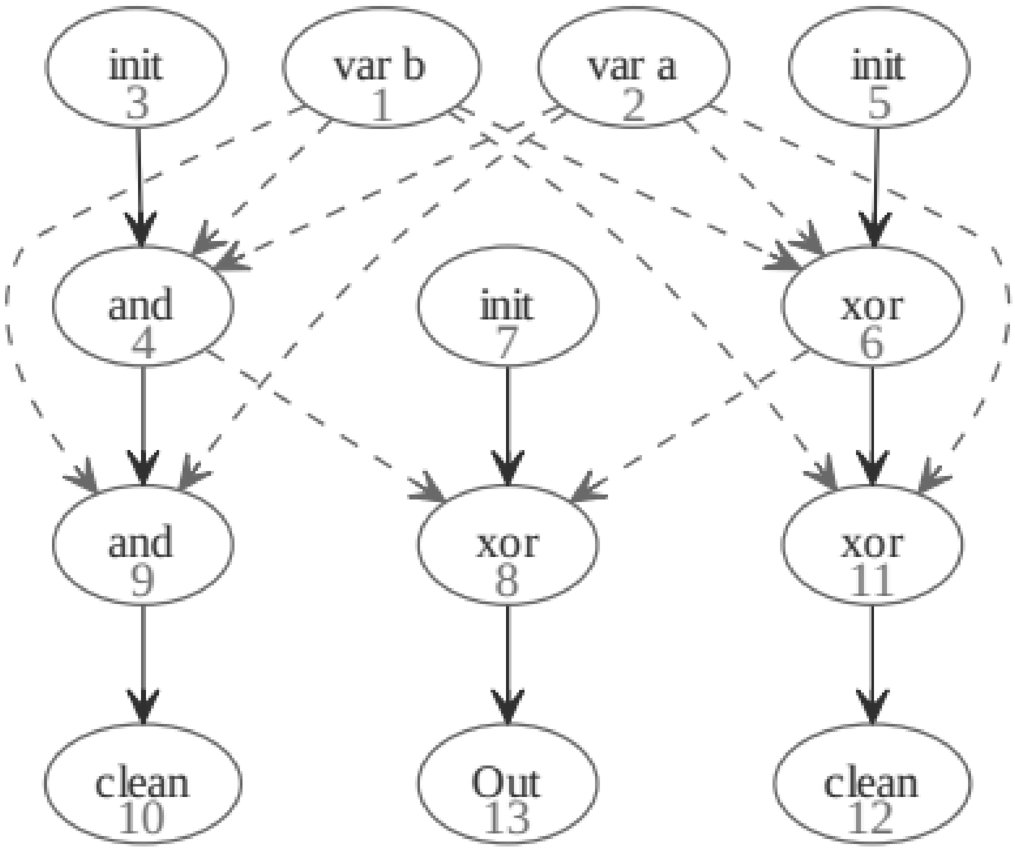


Mutable data dependency graph (MDD)

Example: function inlining; Boolean ops

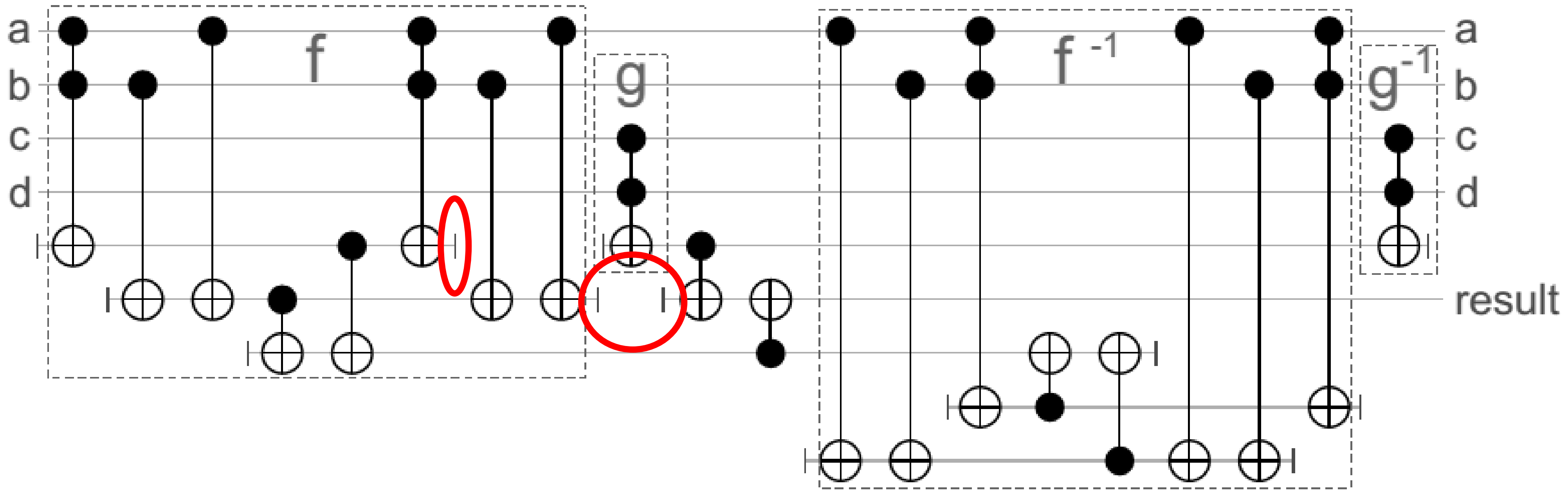
```
let f a b =
    a || b
let g a b =
    a && b
let h a b c d =
    f a b <> g c d
```

Corresponding MDD (only graph for f is shown; similar for g, h)



Example (cont'd)

Generated reversible circuit



Note: - all ancilla qubits (scratch bits) are returned back in the 0 state (indicated by "|")
- Some ancilla qubits are reused in the circuit (red circles above)
- Leads to space savings and offers advantage over alternative methods (e.g. original Bennett)

Algorithm to clean up qubits early

Algorithm 2 EAGER Performs eager clean-up of an MDD.

Require: An MDD G in reverse topological order, subroutines LastDependentNode, ModificationPath.

```
i ← 0
for each node in G do
    if modificationArrows node = ∅ then
        dIndex ← LastDependentNode of node in G
        path ← ModificationPath of node in G
        input ← InputNodes of path in G
        if None (modificationArrows input) ≥ dIndex then
            cleanUp ← (Reverse path) ++ cleanNode
        end if
    else
        cleanUp ← uncleanNode
        G ← Insert cleanUp Into G After dIndex
    end if
end for
return G
```

REVS: Examples

An example at scale: SHA-2

Hash function:

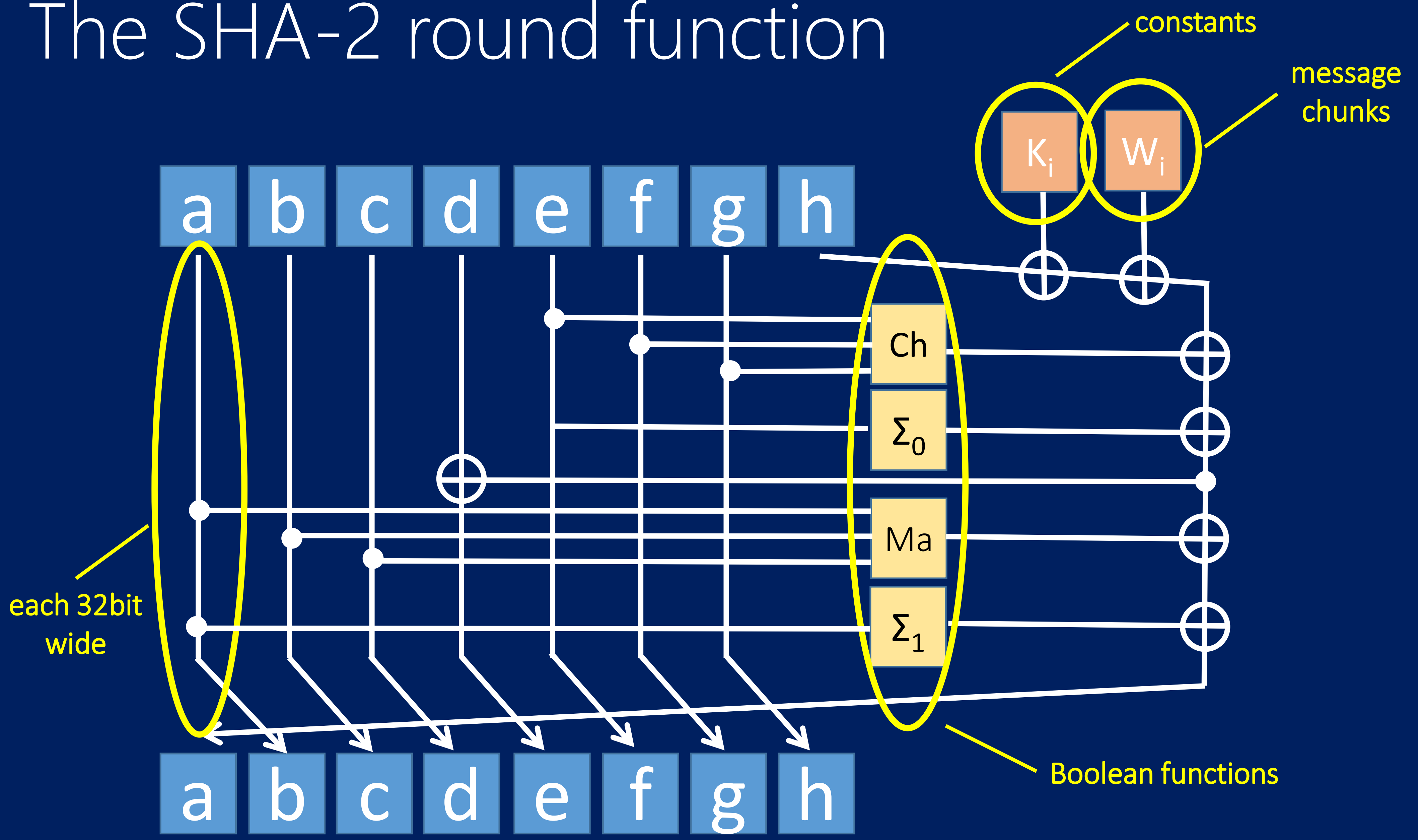
```
Initialize hash values
h0  := 0x6a09e667
h1  := 0xbb67ae85
…
h7  := 0x5be0cd19
Initialize constants
k[0..63] := 0x428a2f98, 0x71374491, 0xb5c0fbcf, …
Do preprocessing
break message into 512-bit chunks (16 32bit ints)
Expand to 64 32 bit ints as follows:
Create W: a 64 entry array of 32 bit ints
Copy the massage into w[0..15] and do:
for each chunk
          for i from 16 to 63
                    s0 := (w[i-15] ≫ 7) ⊕ (w[i-15] ≫ 18) ⊕ (w[i-15] ≫ 3)
                    s1 := (w[i-2] ≫17) ⊕ (w[i-2] ≫19) ⊕ (w[i-2] rshift 10)
                    w[i] := w[i-16] + s0 + w[i-7] + s1
          Initialize working variables to current hash value:
          a := h0
          …
          h := h7 Compression function main loop:
          Do compression rounds
          Add the compressed chunk to the current hash value:
          h0 := h0 + a
          …
          h7 := h7 + h
digest := hash := h0 :: h1 :: h2 :: h3 :: h4 :: h5 :: h6 :: h7
```


[Source: Wikipedia]

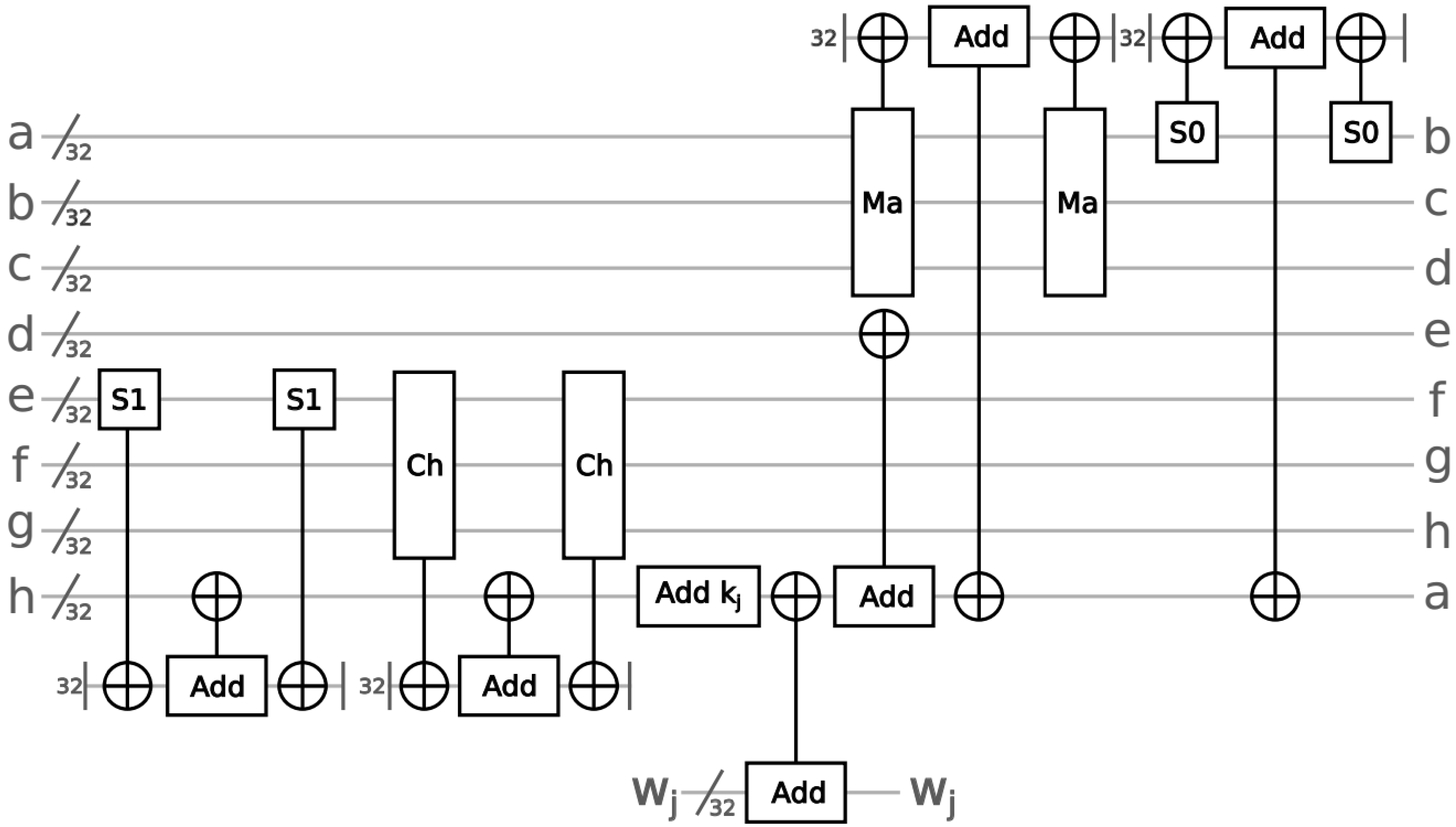
Example: SHA-2 (in F#)

```
let hash x =
  let a = x.[0..31], b = x.[32..63], c = x.[64..95],
    d = x.[96..127], e = x.[128..159], f = x.[160..191],
    g = x.[192..223], h = x.[224..255]
  (%modAdd 32) (ch e f g) h
  (%modAdd 32) (s0 a) h
  (%modAdd 32) w h
  (%modAdd 32) k h
  (%modAdd 32) h d
  (%modAdd 32) (ma a b c) h
  (%modAdd 32) (s1 e) h
for i in 0 .. n - 1 do
  hash (rot 32*i x)
```

The SHA-2 round function



SHA-2: hand-optimized reversible circuit



SHA-2: comparing different cleanup methods

	Bennett cleanup			Eager cleanup			Hand Optimized	
Rounds	#qubits	Toffoli count	time	#qubits	Toffoli count	time	#qubits	Toffoli count
1	704	1124	0.2546002	353	690	0.3290822	353	683
2	832	2248	0.2639522	353	1380	0.3360352	353	1366
3	960	3372	0.2823012	353	2070	0.3420732	353	2049
4	1088	4496	0.2827132	353	2760	0.3543582	353	2732
5	1216	5620	0.2907102	353	3450	0.3664272	353	3415
6	1344	6744	0.3042492	353	4140	0.3784522	353	4098
7	1472	7868	0.3123962	353	4830	0.3918812	353	4781
8	1600	8992	0.3284542	353	5520	0.4025412	353	5464
9	1728	10116	0.3341342	353	6210	0.4130702	353	6147
10	1856	11240	0.3449002	353	6900	0.4304762	353	6830

All timings measured running the F# compiler in VS 2013 on an Intel i7-3667 @ 2Ghz 8GB RAM (6 cores) under Win 8.1

We're beating many REVLIB benchmarks

	Our Method			RevLib		Comparison (rel.)		
name	Tot. Bits	Ancillas	Toffolis	Tot. Bits	Toffolis	Tot. Bits	Toffolis	Time
4mod5	7	2	1	7	4	1.00	0.25	0.00s
5xp1	23	6	83	23	365	1.00	0.23	0.02s
6sym	11	4	35	14	16	0.79	2.19	0.02s
alu4	61	39	2821	33	10456	1.85	0.27	3.70s
apex5	228	23	3727	1025	1860	0.22	2.00	15.59s
bw	36	3	73	87	159	0.41	0.46	0.01s
con1	13	4	16	13	63	1.00	0.25	0.01s
decod24	6	0	1	6	4	1.00	0.25	0.00s
e64	193	63	4096	195	130	0.99	31.5	0.17s
ex1010	38	18	6581	29	31219	1.31	0.21	6.92s
f51m	52	30	1774	35	6207	1.49	0.29	1.97s
frg2	336	54	8950	1219	2186	0.28	4.09	1913.09s
hwb9	33	15	2915	170	394	0.19	7.40	3.13s
max46	20	10	195	17	689	1.18	0.28	0.20s
mini-alu	9	3	14	10	10	0.90	1.40	0.00s
pdc	102	46	3222	619	1105	0.16	2.91	85.16s
rd84	26	14	170	34	50	0.76	3.40	0.13s
seq	107	31	3310	1617	3343	0.07	0.99	1.21s
spla	95	33	3232	489	1054	0.19	3.07	75.11s
sqrt8	18	6	32	18	158	1.00	0.20	0.02s
squar5	16	3	36	17	155	0.94	0.23	0.01s
t481	19	2	26	20	68	0.95	0.38	0.01s

Bold = we beat in size + width

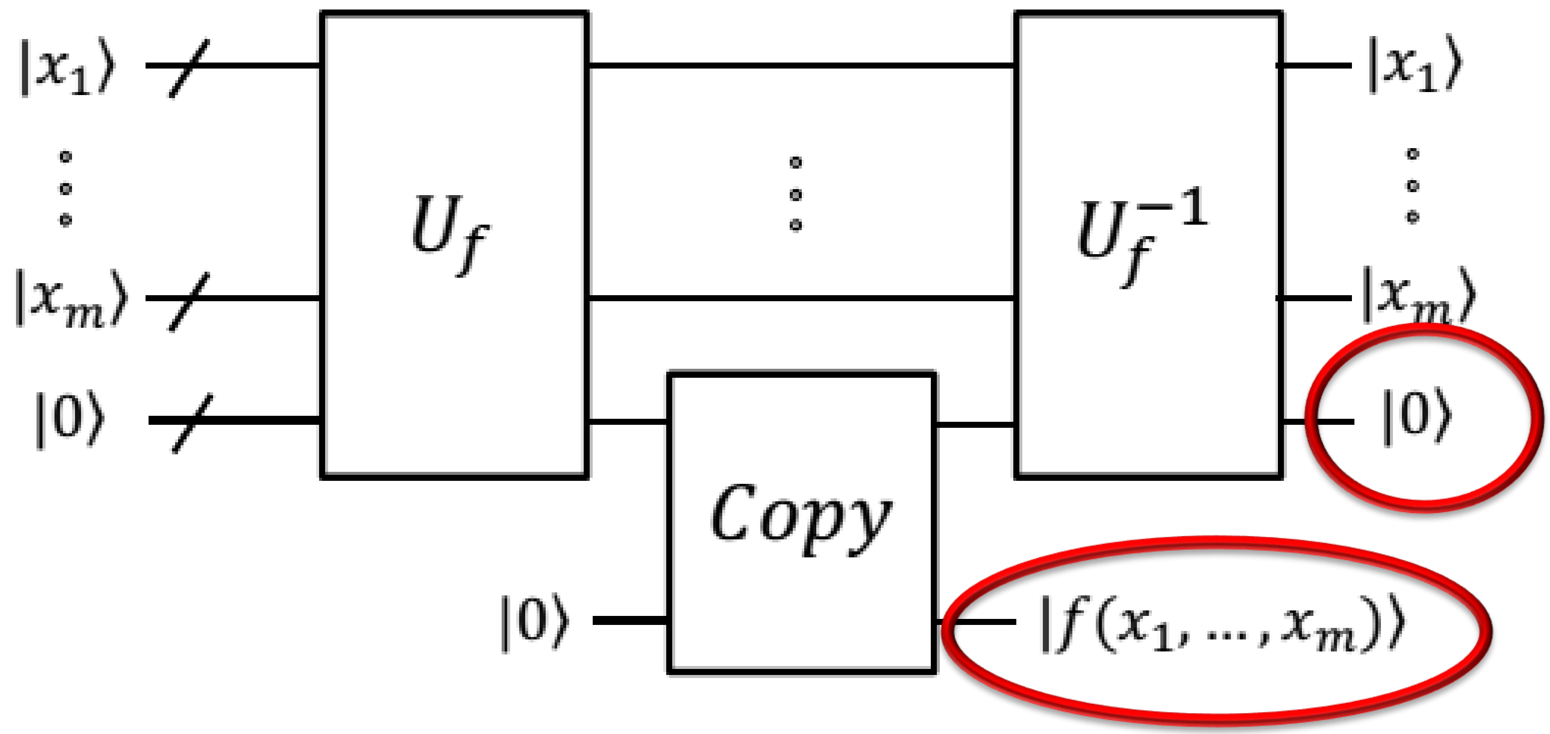
Normal = we beat in width

Simulating Toffoli networks is easy

```
type Primitive =
    | RTOFF of int * int * int
    | RCNOT of int * int
    | RNOT of int

let simCircuit (gates:Primitive list) (numberOfBits:int) (input:bool list) =
    let bits = Array.init numberOfBits (fun _ -> false)
    List.iteri (fun i elm -> bits.[i] <- elm) input
    let applyGate gate =
        match gate with
        | RNOT a -> bits.[a] <- not bits.[a]
        | RCNOT(a, b) -> bits.[b] <- bits.[b] <> bits.[a]
        | RTOFF(a, b, c) -> bits.[c] <- bits.[c] <> (bits.[a] && bits.[b])
    List.iter applyGate gates
    bits
```

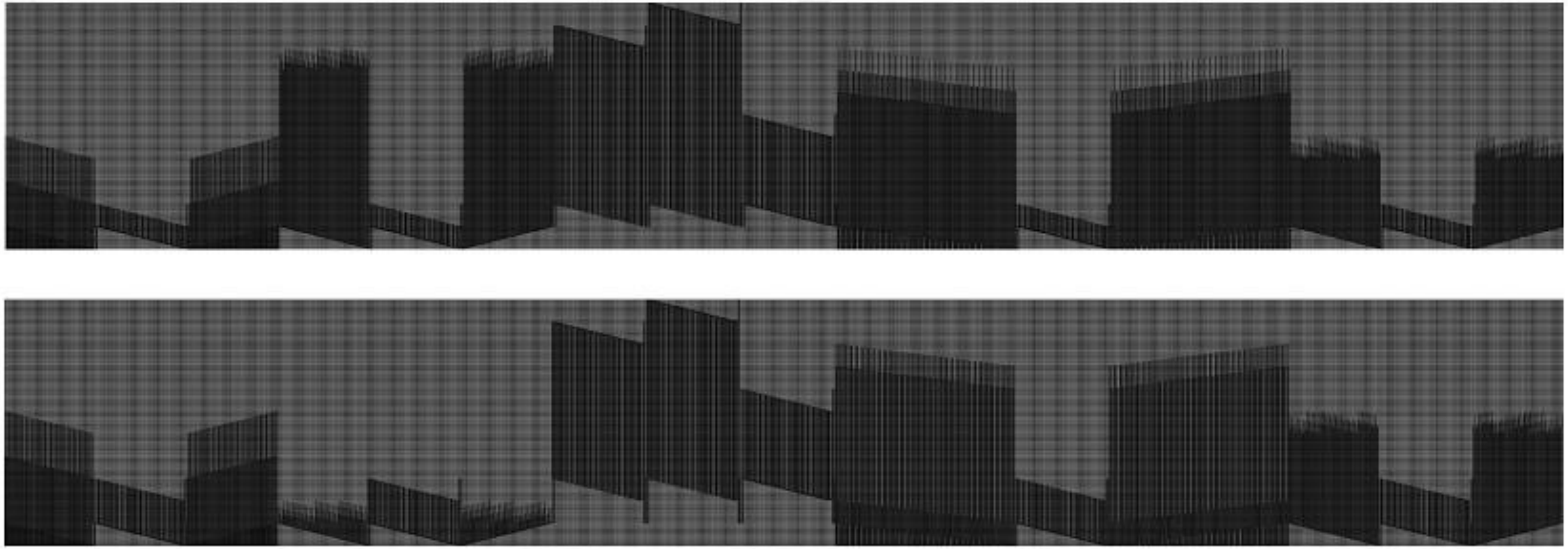
Compiler verification

Why verify?



How do we know that these are indeed the outputs of the circuit?

Simulating Toffoli networks is easy



Reversible Toffoli network computing (?) a SHA-2 hash function
with 353 bits, 3334 gates
Generated by Revs & rendered by LIQui|⟩

ReVer

An irreversible program to reversible circuit compiler, implemented and verified in F (https://www.fstar-lang.org/)*

What that does mean:

- The program interpreter and compiled circuit produce the same output
- Compiled circuits return all ancillas to their initial state

What that doesn't mean:

- That the compiled program is correct
- That the F* proof checker is correct
- The the compiled circuit will produce the same output for every interpreter/hardware

ReVer: Operational semantics

Store $\sigma : \mathbb{N} \rightharpoonup \mathbb{B}$

Config $c ::= \langle t, \sigma \rangle$

$$[\text{REFL}]\ \frac{}{\langle v, \sigma \rangle \Rightarrow \langle v, \sigma \rangle}$$

$$[\text{LET}]\ \frac{\langle t_1, \sigma \rangle \Rightarrow \langle v_1, \sigma' \rangle \qquad \langle t_2[x \mapsto v_1], \sigma' \rangle \Rightarrow \langle v_2, \sigma'' \rangle}{\langle \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2, \sigma \rangle \Rightarrow \langle v_2, \sigma'' \rangle}$$

$$[\text{APP}]\ \frac{\langle t_1, \sigma \rangle \Rightarrow \langle \lambda x.t_1', \sigma' \rangle \qquad \langle t_2, \sigma' \rangle \Rightarrow \langle v_2, \sigma'' \rangle \qquad \langle t_1'[x \mapsto v_2], \sigma'' \rangle \Rightarrow \langle v, \sigma''' \rangle}{\langle (t_1\ t_2), \sigma \rangle \Rightarrow \langle v, \sigma''' \rangle}$$

$$[\text{SEQ}]\ \frac{\langle t_1, \sigma \rangle \Rightarrow \langle \mathsf{unit}, \sigma' \rangle \qquad \langle t_2, \sigma' \rangle \Rightarrow \langle v, \sigma'' \rangle}{\langle t_1; t_2, \sigma \rangle \Rightarrow \langle v, \sigma'' \rangle}$$

$$[\text{ASSN}]\ \frac{\langle t_1, \sigma \rangle \Rightarrow \langle l_1, \sigma' \rangle \qquad \langle t_2, \sigma' \rangle \Rightarrow \langle l_2, \sigma'' \rangle}{\langle t_1 \leftarrow t_2, \sigma \rangle \Rightarrow \langle \mathsf{unit}, \sigma''[l_1 \mapsto \sigma''(l_2)] \rangle}$$

$$[\text{BEXP}]\ \frac{\langle t_1, \sigma \rangle \Rightarrow \langle l_1, \sigma' \rangle \qquad \langle t_2, \sigma' \rangle \Rightarrow \langle l_2, \sigma'' \rangle \qquad l_3 \notin \mathsf{dom}(\sigma'')}{\langle t_1 \star t_2, \sigma \rangle \Rightarrow \langle l_3, \sigma''[l_3 \mapsto \sigma''(l_1) \star \sigma''(l_2)] \rangle}$$

$$[\text{TRUE}]\ \frac{l \notin \mathsf{dom}(\sigma)}{\langle \mathsf{true}, \sigma \rangle \Rightarrow \langle l, \sigma''[l \mapsto 1] \rangle}$$

$$[\text{FALSE}]\ \frac{l \notin \mathsf{dom}(\sigma)}{\langle \mathsf{false}, \sigma \rangle \Rightarrow \langle l, \sigma''[l \mapsto 0] \rangle}$$

$$[\text{APPEND}]\ \frac{\langle t_1, \sigma \rangle \Rightarrow \langle \mathsf{register}\ l_1 \dots l_m, \sigma' \rangle \qquad \langle t_2, \sigma' \rangle \Rightarrow \langle \mathsf{register}\ l_{m+1} \dots l_n, \sigma'' \rangle}{\langle \mathsf{append}\ t_1\ t_2, \sigma \rangle \Rightarrow \langle \mathsf{register}\ l_1 \dots l_n, \sigma'' \rangle}$$

$$[\text{INDEX}]\ \frac{\langle t, \sigma \rangle \Rightarrow \langle \mathsf{register}\ l_1 \dots l_n, \sigma' \rangle \qquad 1 \le i \le n}{\langle t.[i], \sigma \rangle \Rightarrow \langle l_i, \sigma' \rangle}$$

$$[\text{SLICE}]\ \frac{\langle t, \sigma \rangle \Rightarrow \langle \mathsf{register}\ l_1 \dots l_n, \sigma' \rangle \qquad 1 \le i \le j \le n}{\langle t.[i..j], \sigma \rangle \Rightarrow \langle \mathsf{register}\ l_i \dots l_j, \sigma' \rangle}$$

$$[\text{REG}]\ \frac{\langle t_1, \sigma \rangle \Rightarrow \langle l_1, \sigma_1 \rangle \quad \langle t_2, \sigma \rangle \Rightarrow \langle l_2, \sigma_2 \rangle \quad \vdots \quad \langle t_n, \sigma \rangle \Rightarrow \langle l_n, \sigma_n \rangle}{\langle \mathsf{register}\ t_1 \dots t_n, \sigma \rangle \Rightarrow \langle \mathsf{register}\ l_1 \dots l_n, \sigma_n \rangle}$$

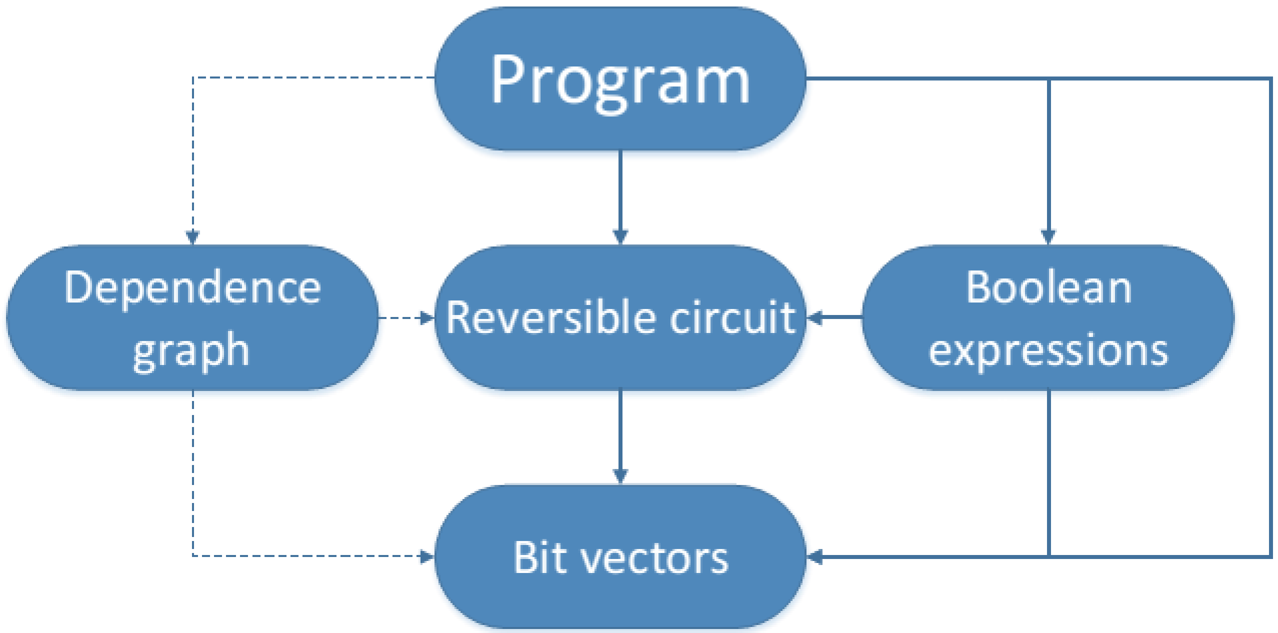
$$[\text{ROTATE}]\ \frac{\langle t, \sigma \rangle \Rightarrow \langle \mathsf{register}\ l_1 \dots l_n, \sigma' \rangle \qquad 1 < i < n}{\langle \mathsf{rotate}\ t\ i, \sigma \rangle \Rightarrow \langle \mathsf{register}\ l_i \dots l_{i-1}, \sigma' \rangle}$$

$$[\text{CLEAN}]\ \frac{\langle t, \sigma \rangle \Rightarrow \langle l, \sigma' \rangle \qquad \sigma'(l) = \mathsf{false}}{\langle \mathsf{clean}\ t, \sigma \rangle \Rightarrow \langle \mathsf{unit}, \sigma'|_{\mathsf{dom}(\sigma') \setminus \{l\}} \rangle}$$

$$[\text{ASSERT}]\ \frac{\langle t, \sigma \rangle \Rightarrow \langle l, \sigma' \rangle \qquad \sigma'(l) = \mathsf{true}}{\langle \mathsf{assert}\ t, \sigma \rangle \Rightarrow \langle \mathsf{unit}, \sigma' \rangle}$$

ReVer architecture overview

Circuit compiler and interpreter. Written and verified in F*



Two verified paths:

- Bennett-style compilation, translate directly to circuit
- Space-efficient Boolean expression compilation

THANK YOU!

http://research.microsoft.com/groups/quarc/

http://research.microsoft.com/en-us/labs/stationq/

$\mathrm{LIQ}Ui|\rangle$ is publicly available from
http://stationq.github.io/Liquid

Microsoft

martinro@microsoft.com