Programming a Quantum Annealer

Quantum Computing Seminar
North Carolina State University

Scott Pakin

30 January 2018

Los Alamos NATIONAL LABORATORY EST.1943

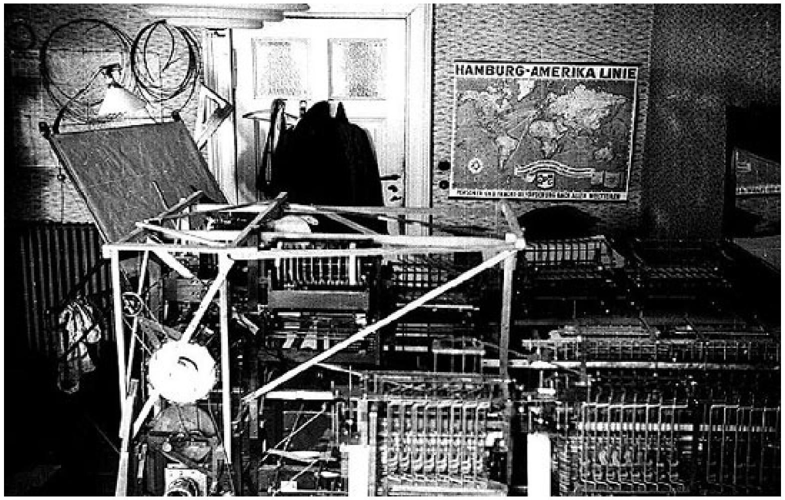
Operated by Los Alamos National Security, LLC for the U.S. Department of Energy's NNSA

LA-UR-18-20727

Outline

- How do you program a quantum annealer?
- Can we do better?
- What problems can you solve?
- What should you learn from all this?

Reminder #1: We’re in the *Very* Early Days of QC



- **Zuse Z1**
  - Completed 1938
  - 1408 bits of memory
  - 8 instruction types
  - 1 Hz clock
  - 1 kW
  - 1000 kg
  - Powered by a vacuum-cleaner motor
  - Programmed in machine language

- **D-Wave 2X**
  - Completed 2015
  - 1152 qubits (nominal)
  - 1 instruction type
  - 200 kHz sampling rate
  - 25 kW
  - 3800 kg
  - Processor kept in a near-vacuum
  - Programmed in machine language (normally, but this talk changes that)

Reminder #2: Quantum Annealers are Special-Purpose Devices

- **Solve a single problem**
  - Find $\arg\min_{\sigma} \mathcal{H}$ with

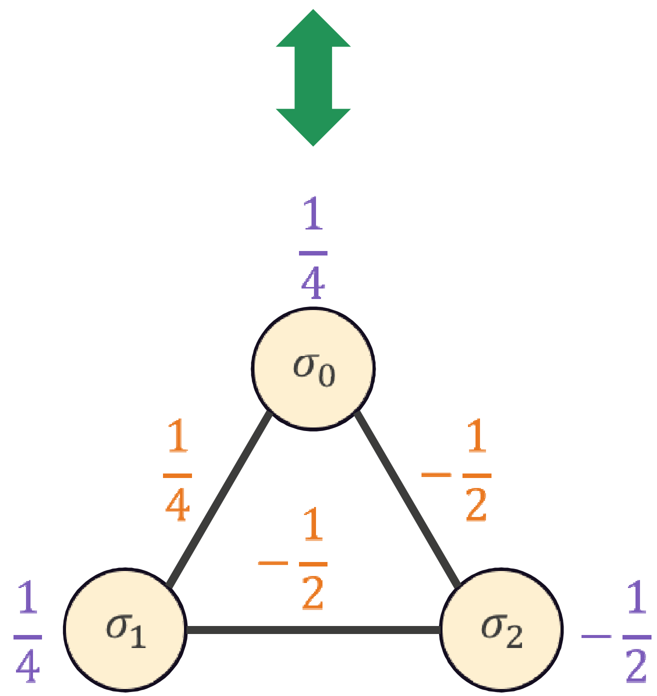
$$\mathcal{H} = \sum_{i=0}^{N-1} h_i \sigma_i + \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} J_{i,j} \sigma_i \sigma_j$$

  - given $h_i \in \mathbb{R}$ and $J_{i,j} \in \mathbb{R}$ and solving for $\sigma_i \in \{-1, +1\}$
- **This is a classical Hamiltonian**
  - All real-valued coefficients
  - Quantum effects are used under the covers to more effectively discover the minima
- **Fundamentally stochastic**
  - No guarantee of receiving the same answer on every run
  - No guarantee of receiving a *correct* answer on any run

Visualizing a Hamiltonian as a Graph

- **Linear terms as vertex weights**
- **Quadratic terms as edge weights**

$$\mathcal{H} = \frac{1}{4}\sigma_0 + \frac{1}{4}\sigma_1 - \frac{1}{2}\sigma_2 + \frac{1}{4}\sigma_0\sigma_1 - \frac{1}{2}\sigma_0\sigma_2 - \frac{1}{2}\sigma_1\sigma_2$$



Alternative Formulation—with Booleans

- **Different names for this appear in the optimization literature**
  - QUBO (quadratic unconstrained binary optimization problem)
  - UBQP (unconstrained binary quadratic optimization problem)
- **Goal**
  - Find $\arg\min_x f(x)$ with

$$f(x) = x^T Q x$$

given Q either symmetric or upper-triangular, $Q_{i,j} \in \mathbb{R}$, and solving for $x_i \in \{0,1\}$

- **Can easily map between Ising-model Hamiltonians and QUBOs**
  - Diagonal elements of Q correspond to h_i; off-diagonal elements correspond to $J_{i,j}$
  - Based on a simple linear transformation: $x_i = (\sigma_i + 1)/2$
  - Hint: $x_i^2 = x_i$ when $x_i \in \{0,1\}$
  - Formula: $Q_{i,j} = 4J_{i,j}$ for $i < j$ and $Q_{i,i} = 2(h_i - \sum_{j=0}^{i-1} J_{j,i} - \sum_{j=i+1}^{N-1} J_{i,j})$
  - Example:

$$\mathcal{H} = \frac{1}{4}\sigma_0 + \frac{1}{4}\sigma_1 - \frac{1}{2}\sigma_2 + \frac{1}{4}\sigma_0\sigma_1 - \frac{1}{2}\sigma_0\sigma_2 - \frac{1}{2}\sigma_1\sigma_2 \quad \overset{\mp\frac{3}{4}}{\Longleftrightarrow} \quad f(x) = x^T \begin{pmatrix} 1 & 1 & -2 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{pmatrix} x$$

(Use $(Q + Q^T)/2$ if you prefer a symmetric matrix.)

Solving a Map-Coloring Problem

- **Given a planar map, color each region with one of four colors such that no two adjacent regions have the same color**
  - NP-hard problem
- **We start by defining a region as having exactly one color**
  - Let's use a unary encoding with +1 ≡ has the color and –1 ≡ lacks the color

σ_{red}	σ_{yellow}	σ_{green}	σ_{blue}
–1	–1	–1	–1
–1	–1	–1	+1
–1	–1	+1	–1
–1	–1	+1	+1
–1	+1	–1	–1
–1	+1	–1	+1
–1	+1	+1	–1
–1	+1	+1	+1

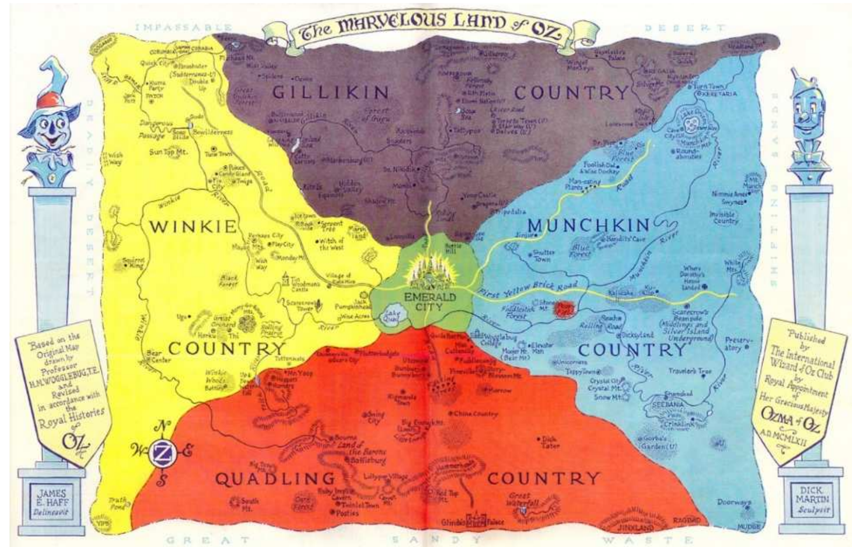
σ_{red}	σ_{yellow}	σ_{green}	σ_{blue}
+1	–1	–1	–1
+1	–1	–1	+1
+1	–1	+1	–1
+1	–1	+1	+1
+1	+1	–1	–1
+1	+1	–1	+1
+1	+1	+1	–1
+1	+1	+1	+1

A Hamiltonian for a Region of a Map

- **Define a system of inequalities**
- **Ground state (four-way degenerate)**
  - $\mathcal{H}(---+) = \mathcal{H}(--+-) = \mathcal{H}(-+--) = \mathcal{H}(+---) = k$
- **All excited states**
  - $\mathcal{H}(----) > k$
  - $\mathcal{H}(-++-) > k$
  - $\mathcal{H}(+-+-) > k$
  - $\mathcal{H}(++-+) > k$
  - $\mathcal{H}(--++) > k$
  - $\mathcal{H}(-+++) > k$
  - $\mathcal{H}(+-++) > k$
  - $\mathcal{H}(+++-) > k$
  - $\mathcal{H}(-+-+) > k$
  - $\mathcal{H}(+--+) > k$
  - $\mathcal{H}(++--) > k$
  - $\mathcal{H}(++++) > k$
- **Expand the Hamiltonian function out to $N = 4$:**
  - $\mathcal{H}(\sigma_r, \sigma_y, \sigma_g, \sigma_b) = h_r\sigma_r + h_y\sigma_y + h_g\sigma_g + h_b\sigma_b + J_{r,y}\sigma_r\sigma_y + J_{r,g}\sigma_r\sigma_g + J_{r,b}\sigma_r\sigma_b + J_{y,g}\sigma_y\sigma_g + J_{y,b}\sigma_y\sigma_b + J_{g,b}\sigma_g\sigma_b$
- **Solve the system of inequalities for the h_i and $J_{i,j}$ coefficients**
  - Opposite of what a quantum annealer does
- **One possible solution (not unique)**
  - $\mathcal{H}(\sigma_r, \sigma_y, \sigma_g, \sigma_b) = \sigma_r + \sigma_y + \sigma_g + \sigma_b + \frac{1}{2}\sigma_r\sigma_y + \frac{1}{2}\sigma_r\sigma_g + \frac{1}{2}\sigma_r\sigma_b + \frac{1}{2}\sigma_y\sigma_g + \frac{1}{2}\sigma_y\sigma_b + \frac{1}{2}\sigma_g\sigma_b$

A Hamiltonian for the Complete Map-Coloring Problem

- **Hamiltonians are additive**
  - We can add up a bunch of region Hamiltonians to produce a map Hamiltonian
- **Use antiferromagnetic couplings ($J_{i,j} > 0$) to avoid assigning adjacent regions the same color***
  - $\sigma_{r_{WC}}\sigma_{r_{GC}} + \sigma_{y_{WC}}\sigma_{y_{GC}} + \sigma_{g_{WC}}\sigma_{g_{GC}} + \sigma_{b_{WC}}\sigma_{b_{GC}} + \sigma_{r_{EC}}\sigma_{r_{MC}} + \sigma_{g_{EC}}\sigma_{g_{MC}} + \cdots$



$\mathcal{H}(\sigma_{r_{GC}}, \sigma_{y_{GC}}, \sigma_{g_{GC}}, \sigma_{b_{GC}})$

$\mathcal{H}(\sigma_{r_{WC}}, \sigma_{y_{WC}}, \sigma_{g_{WC}}, \sigma_{b_{WC}})$ | $\mathcal{H}(\sigma_{r_{EC}}, \sigma_{y_{EC}}, \sigma_{g_{EC}}, \sigma_{b_{EC}})$ | $\mathcal{H}(\sigma_{r_{MC}}, \sigma_{y_{MC}}, \sigma_{g_{MC}}, \sigma_{b_{MC}})$

$\mathcal{H}(\sigma_{r_{QC}}, \sigma_{y_{QC}}, \sigma_{g_{QC}}, \sigma_{b_{QC}})$

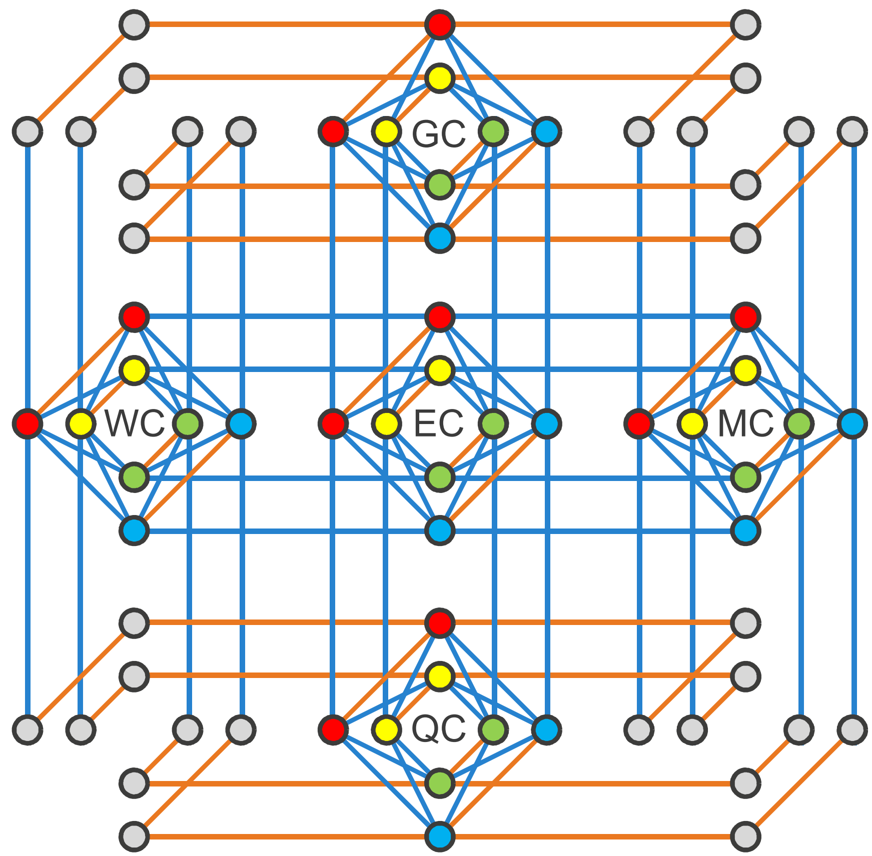
*Oversimplification: OK if neither of two adjacent regions has a given color. Adding $\mathcal{H} = \sigma_i + \sigma_j + \sigma_i\sigma_j$ should do the trick.

Embedding the Problem in a Chimera Graph

- **Each qubit in a region needs to couple with all three other qubits**

  and

- **EC needs to be able to couple to the north (GC), south (QC), east (MC), and west (WC)**
  - Solution: Replace each qubit with two ferromagnetically coupled ($J_{i,j} < 0$) qubits
  - One qubit couples north/south and one qubit couples east/west
- **All regions except EC need to be able to couple diagonally**
  - Solution: Introduce "ghost" unit cells solely for routing
  - Alternative: Replicate regions (two unit cells for each region but EC) and couple ferromagnetically

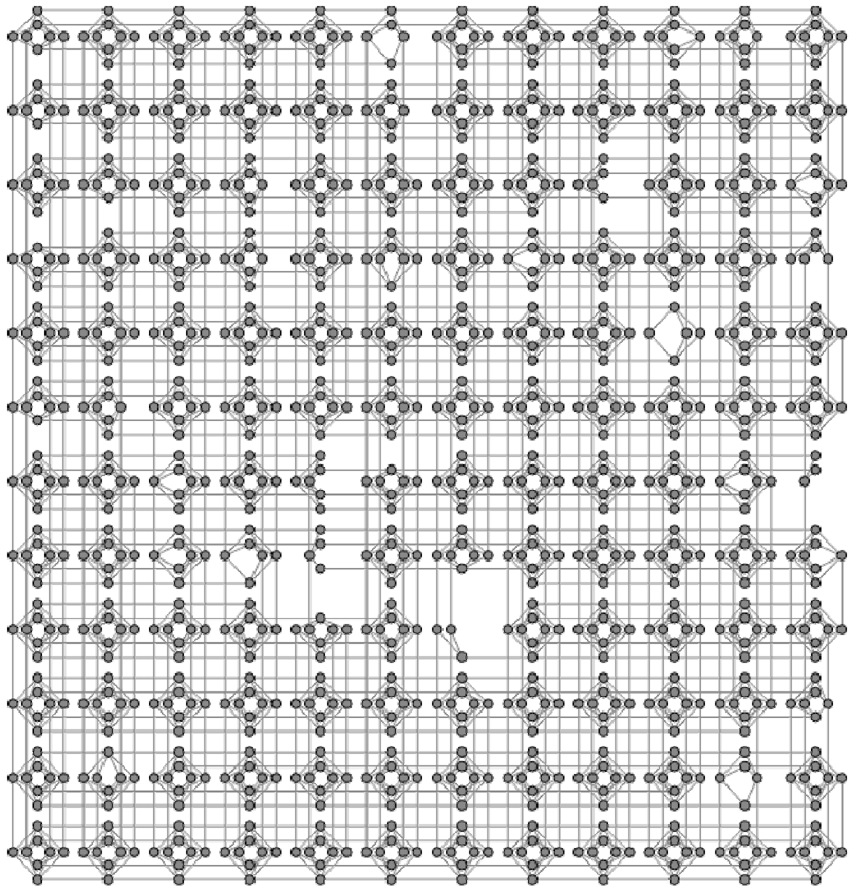


Outline

- How do you program a quantum annealer?
- **Can we do better?**
- What problems can you solve?
- What should you learn from all this?

Goal

- **Compile an ordinary(-ish) classical program to a 2-local Ising-model Hamiltonian, $\mathcal{H} = \sum_{i=0}^{N-1} h_i \sigma_i + \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} J_{i,j} \sigma_i \sigma_j$, such that $\arg\min_{\sigma} \mathcal{H}$ corresponds to a valid mapping of program inputs to outputs**
  - In fact, we need to compile to the physical Hamiltonian implemented by the hardware (a subgraph of a Chimera graph)
- **Is this even possible?**
  - Yes!
  - Over the next few slides we'll consider higher and higher levels of abstraction until we achieve our goal



Physical topology of LANL's D-Wave 2X system, Ising (1095 active qubits out of a nominal 1152)

Interpreting the Problem Hamiltonian

- **Let's start by considering only the external field (the h_i values):**

$$\mathcal{H} = \sum_{i=0}^{N-1} h_i \sigma_i + \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} J_{i,j} \sigma_i \sigma_j$$

- **We arbitrarily call $\sigma_i = +1$ "TRUE" and $\sigma_i = -1$ "FALSE"**
- **Here are the optimal values of σ_i for different values of h_i:**

Negative (say, $h_i = -5$)

σ_i	$h_i \sigma_i$
–1	+5
+1	–5

Zero

σ_i	$h_i \sigma_i$
–1	0
+1	0

Positive (say, $h_i = +5$)

σ_i	$h_i \sigma_i$
–1	–5
+1	+5

- **Observations**
  - A negative h_i means, "I want σ_i to be TRUE"
  - A zero h_i means, "I don't care if σ_i is TRUE or FALSE"
  - A positive h_i means, "I want σ_i to be FALSE"

Interpreting the Problem Hamiltonian (cont.)

- **Now let's consider only the coupler strengths (the $J_{i,j}$ values):**

$$\mathcal{H} = \sum_{i=0}^{N-1} h_i \sigma_i + \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} J_{i,j} \sigma_i \sigma_j$$

- **Here are the optimal values of σ_i and σ_j for different values of $J_{i,j}$:**

Negative ($J_{i,j} = -5$)

σ_i	σ_j	$J_{i,j}\sigma_i\sigma_j$
–1	–1	–5
–1	+1	+5
+1	–1	+5
+1	+1	–5

Zero

σ_i	σ_j	$J_{i,j}\sigma_i\sigma_j$
–1	–1	0
–1	+1	0
+1	–1	0
+1	+1	0

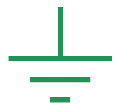
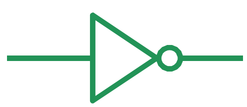
Positive ($J_{i,j} = +5$)

σ_i	σ_j	$J_{i,j}\sigma_i\sigma_j$
–1	–1	+5
–1	+1	–5
+1	–1	–5
+1	+1	+5

- **Observations**
  - A negative $J_{i,j}$ means, "I want σ_i and σ_j to be equal"
  - A zero $J_{i,j}$ means, "I don't care how σ_i and σ_j are related"
  - A positive $J_{i,j}$ means, "I want σ_i and σ_j to be different"

Interpretation

- **Look what we can express as Hamiltonians so far:**

Component	Hamiltonian
 ground	$\mathcal{H}_{\mathrm{GND}} = \sigma_g$
V_{CC}  power	$\mathcal{H}_{\mathrm{VCC}} = -\sigma_v$
wire	$\mathcal{H}_{\mathrm{wire}} = -\sigma_A \sigma_Y$
 inverter	$\mathcal{H}_{\neg} = \sigma_A \sigma_Y$

Expressing Logic Gates as Hamiltonians

- **Write a *complete* truth table, distinguishing valid from invalid rows**
- **Set up a system of inequalities**
  - All valid rows must evaluate to the same value
  - All invalid rows must evaluate to a value greater than that of any valid row
- **Example: 2-input AND gate ($Y = A \wedge B$)**

σ_A	σ_B	σ_Y	$\mathcal{H}_\wedge(\sigma_A, \sigma_B, \sigma_Y)$	Must be
–1	–1	–1	$-h_A - h_B - h_Y + J_{A,B} + J_{A,Y} + J_{B,Y}$	$= k$
–1	–1	+1	$-h_A - h_B + h_Y + J_{A,B} - J_{A,Y} - J_{B,Y}$	$> k$
–1	+1	–1	$-h_A + h_B - h_Y - J_{A,B} + J_{A,Y} - J_{B,Y}$	$= k$
–1	+1	+1	$-h_A + h_B + h_Y - J_{A,B} - J_{A,Y} + J_{B,Y}$	$> k$
+1	–1	–1	$+h_A - h_B - h_Y - J_{A,B} - J_{A,Y} + J_{B,Y}$	$= k$
+1	–1	+1	$+h_A - h_B + h_Y - J_{A,B} + J_{A,Y} - J_{B,Y}$	$> k$
+1	+1	–1	$+h_A + h_B - h_Y + J_{A,B} - J_{A,Y} - J_{B,Y}$	$> k$
+1	+1	+1	$+h_A + h_B + h_Y + J_{A,B} + J_{A,Y} + J_{B,Y}$	$= k$

Expressing Logic Gates as Hamiltonians (cont.)

- **Problem: Not all *N*-input gates can be expressed with *N*+1 qubits**
  - System of inequalities may be unsolvable
  - Example: 2-input XOR ($Y = A \oplus B$)
- **Solution: Introduce ancilla qubits for more degrees of freedom**
  - Keep same number of valid rows
  - How many ancillas and which rows should be valid? That's an open question.

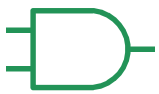
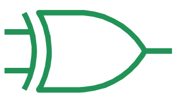
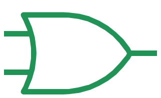
σ_A	σ_B	σ_Y
–1	–1	–1
–1	–1	+1
–1	+1	–1
–1	+1	+1
+1	–1	–1
+1	–1	+1
+1	+1	–1
+1	+1	+1

σ_A	σ_B	σ_Y	σ_a
–1	–1	–1	–1
–1	–1	–1	+1
–1	–1	+1	–1
–1	–1	+1	+1
–1	+1	–1	–1
–1	+1	–1	+1
–1	+1	+1	–1
–1	+1	+1	+1

σ_A	σ_B	σ_Y	σ_a
+1	–1	–1	–1
+1	–1	–1	+1
+1	–1	+1	–1
+1	–1	+1	+1
+1	+1	–1	–1
+1	+1	–1	+1
+1	+1	+1	–1
+1	+1	+1	+1

Increasing our Repertoire

- **We can define Hamiltonians for whatever gates we want**

Gate	Hamiltonian
 AND	$\mathcal{H}_{\wedge} = -\frac{1}{2}\sigma_A - \frac{1}{2}\sigma_B + \sigma_Y + \frac{1}{2}\sigma_A\sigma_B - \sigma_A\sigma_Y - \sigma_B\sigma_Y$
 XOR	$\mathcal{H}_{\oplus} = \frac{1}{2}\sigma_A + \frac{1}{2}\sigma_B + \frac{1}{2}\sigma_Y + \sigma_a + \frac{1}{2}\sigma_A\sigma_B + \frac{1}{2}\sigma_A\sigma_Y + \sigma_A\sigma_a + \frac{1}{2}\sigma_B\sigma_Y + \sigma_B\sigma_a + \sigma_Y\sigma_a$
 OR	$\mathcal{H}_{\vee} = \frac{1}{2}\sigma_A + \frac{1}{2}\sigma_B - \sigma_Y + \frac{1}{2}\sigma_A\sigma_B - \sigma_A\sigma_Y - \sigma_B\sigma_Y$

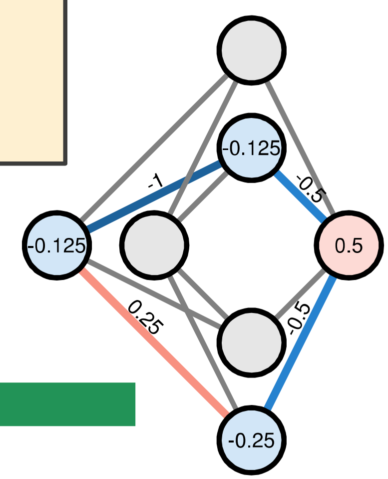
- **Important feature: Hamiltonians can be added**
  - Gate + wire + gate = circuit

A Standard Cell Library

- **Implement using QMASM, my quantum macro assembler**
  - Open-source software, available from https://github.com/lanl/qmasm
- **Symbolic Hamiltonians**
  - QMASM automatically maps user-defined qubit names to physical qubit numbers on a D-Wave system's specific Chimera graph
  - Reports results in terms of qubit names, not numbers
- **Macros**
  - Define reusable components (e.g., gates) that can be instantiated repeatedly
- **Include files**
  - Put collections of macros (e.g., a standard cell library) in a separate file that can be included by multiple programs

$$\mathcal{H}_{\wedge} = -\frac{1}{2}\sigma_A - \frac{1}{2}\sigma_B + \sigma_Y + \frac{1}{2}\sigma_A\sigma_B - \sigma_A\sigma_Y - \sigma_B\sigma_Y$$

```
# Y = A AND B
!begin_macro AND
  A -0.5
  B -0.5
  Y  1

  A B  0.5
  A Y -1
  B Y -1
!end_macro AND
```

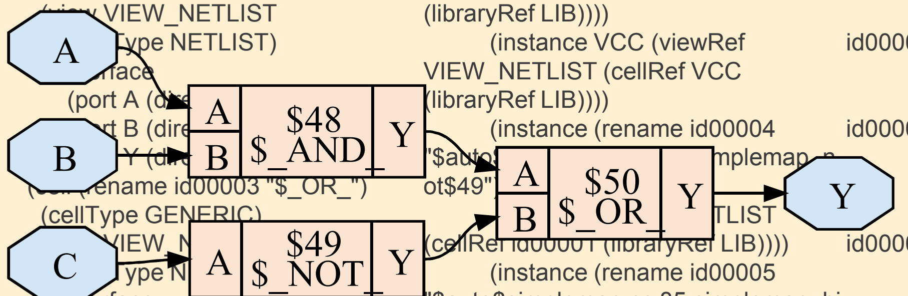


A	B	Y
F	F	F
F	T	F
T	F	F
T	T	T

Hardware Netlists

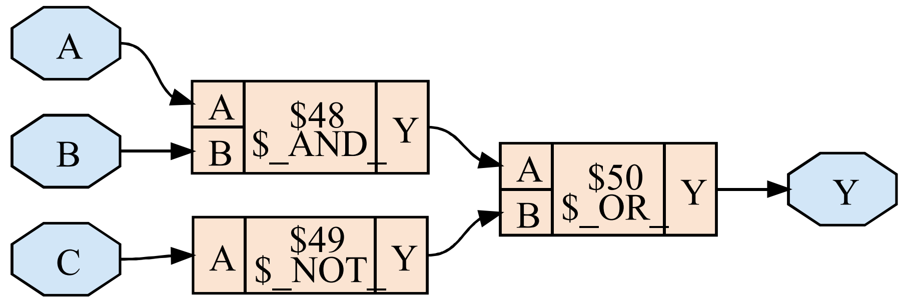
- **Low-level circuit description**
  - Machine-parseable list of gates and wires
- **Semi-standard: EDIF**
  - Electronic Data Interchange Format

```
(edif example
  (edifVersion 2 0 0)
  (edifLevel 0)
  (keywordMap (keywordLevel 0))
  (comment "Generated by Yosys 0.7
(git sha1 61f6811, gcc 6.2.0-11ubuntu1
-O2 -fdebug-prefix-map=/build/yosys-
OIL3SR/yosys-0.7=. -fstack-protector-
strong -fPIC -Os)")
  (external LIB
    (edifLevel 0)
    (technology (numberDefinition))
    (cell GND
      (cellType GENERIC)
      (view VIEW_NETLIST
        (viewType NETLIST)
        (interface (port G (direction
OUTPUT)))))
    (cell VCC
      (cellType GENERIC)
      (view VIEW_NETLIST
        (viewType NETLIST)
        (interface (port P (direction
OUTPUT)))))
    (cell (rename id00001 "$_NOT_")
      (cellType GENERIC)
      (view VIEW_NETLIST
        (viewType NETLIST)
        (interface
          (port A (direction INPUT))
          (port Y (direction OUTPUT)))))
    (cell (rename id00002 "$_AND_")
      (cellType GENERIC)
      (view VIEW_NETLIST
        (viewType NETLIST)
        (interface
          (port A (d
          (port B (dire
          (port Y (d
    (cell (rename id00003 "$_OR_")
      (cellType GENERIC)
      (view VIEW_N
        (viewType N
        (interface
          (port A (direction INPUT))
          (port B (direction INPUT))
          (port Y (direction OUTPUT))))))
  (library DESIGN
    (edifLevel 0)
    (technology (numberDefinition))
    (cell example
      (cellType GENERIC)
      (view VIEW_NETLIST
        (viewType NETLIST)
        (interface
          (port A (direction INPUT))
          (port B (direction INPUT))
          (port C (direction INPUT))
          (port Y (direction OUTPUT)))
        (contents
          (instance GND (viewRef
VIEW_NETLIST (cellRef GND
(libraryRef LIB))))
          (instance VCC (viewRef
VIEW_NETLIST (cellRef VCC
(libraryRef LIB))))
          (instance (rename id00004
"$auto  mplemap_n
ot$49"  TLIST
(cellRef id00001 (libraryRef LIB))))
          (instance (rename id00005
"$auto$simplemap.cc:85:simplemap_bi
top$48")
            (viewRef VIEW_NETLIST
(cellRef id00002 (libraryRef LIB))))
          (instance (rename id00006
"$auto$simplemap.cc:85:simplemap_bi
top$50")
            (viewRef VIEW_NETLIST
(cellRef id00003 (libraryRef LIB))))
          (net (rename id00007
"$and$example.v:4$1_Y") (joined
            (portRef A (instanceRef
id00006))
            (portRef Y (instanceRef
id00005))))
          (net (rename id00008
"$not$example.v:4$2_Y") (joined
            (portRef B (instanceRef
id00006))
            (portRef Y (instanceRef
id00004))))
          (net Y (joined
            (portRef Y (instanceRef
id00006))
            (portRef Y)))
          (net C (joined
            (portRef A (instanceRef
id00004))
            (portRef C)))
          (net A (joined
            (portRef A (instanceRef
id00005))
            (portRef A)))
          (net B (joined
            (portRef B (instanceRef
id00005))
            (portRef B)))))))
  (design example
    (cellRef example (libraryRef
DESIGN))))
```



Conversion to QMASM

- **Implement using edif2qmasm**
  - Open-source software, available from https://github.com/lanl/edif2qmasm
- **Straightforward mapping**
  - *Gates*: EDIF cell instances → QMASM macro instantiations (“**!use_macro**”)
  - *Wires*: EDIF nets → QMASM chains (“=”)
- **We can now run a digital circuit on a D-Wave system!**
- **But how do we generate an EDIF netlist in the first place?**



[EDIF code from previous slide]

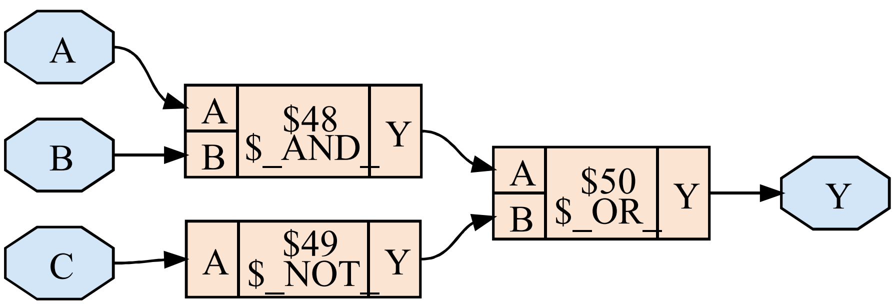
edif2qmasm

```
!include <stdcell>

!begin_macro example
  !use_macro AND $id00005
  !use_macro NOT $id00004
  !use_macro OR $id00006
  $id00004.A = C
  $id00005.A = A
  $id00005.B = B
  $id00006.A = $id00005.Y
  $id00006.B = $id00004.Y
  $id00006.Y = Y
!end_macro example

!use_macro example example
```

Leveraging Decades of Computer Engineering

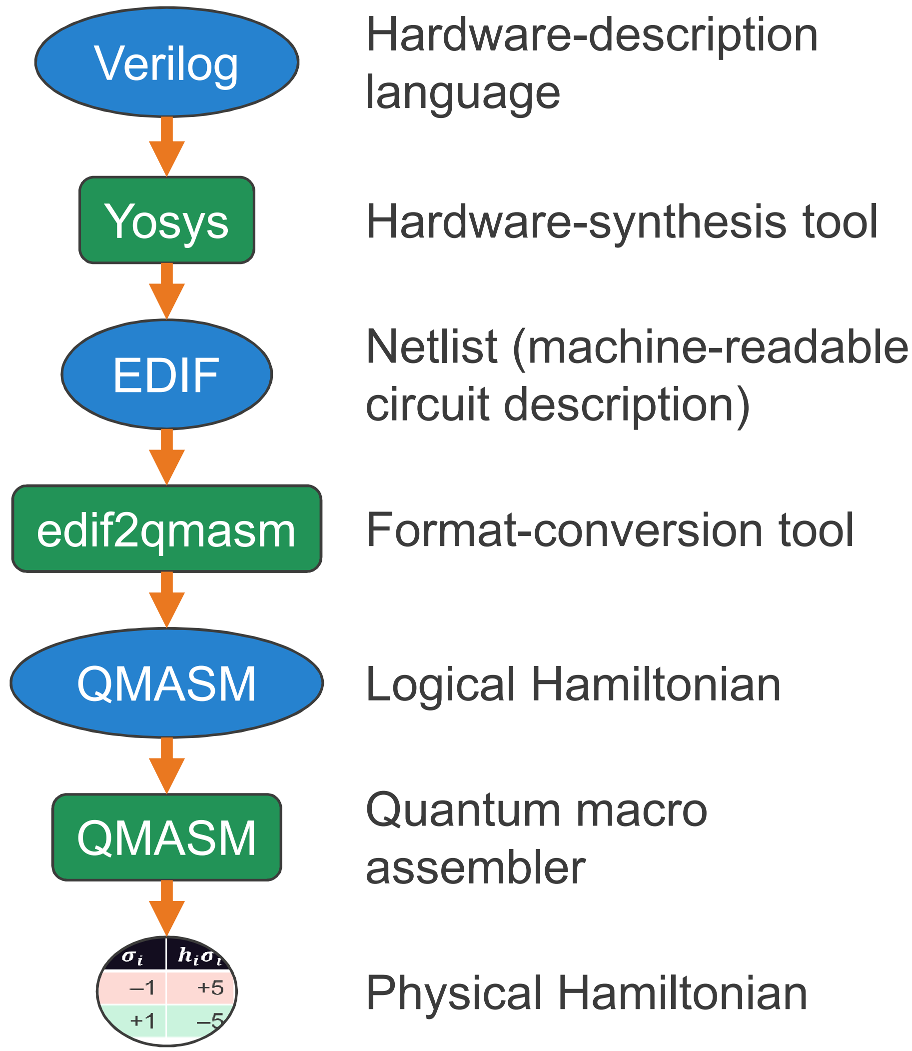
- **Today, virtually all non-trivial hardware is created using a hardware description language (HDL)**
  - Looks more-or-less like an ordinary programming language
  - Variables, arithmetic operators, relational operators, conditionals, loops, modules, …
- **Hardware synthesis tools compile HDLs to a set of logic primitives**
  - AND, OR, NOT, XOR, …
- **Often perform a variety of transformations to reduce the amount of logic required**
- **My toolbox**
  - HDL: Verilog (first introduced in 1984)
  - Hardware synthesis tool: Yosys (https://github.com/cliffordwolf/yosys) with additional optimizations provided by ABC (https://bitbucket.org/alanmi/abc)

```
module example (A, B, C, Y);
  input A, B, C;
  output Y;
  assign Y = (A&B) | ~C;
endmodule
```



Summary of Approach

- **Start with a program written in a hardware-description language**
- **Let an existing hardware-synthesis tool compile the HDL to a circuit of Boolean operators**
- **Convert the circuit to QMASM using edif2qmasm**
- **Generate a D-Wave-specific Ising Hamiltonian from the QMASM code**
- **Run on a D-Wave**

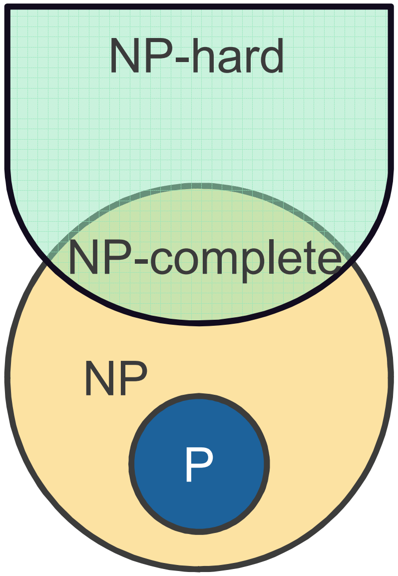


Outline

- How do you program a quantum annealer?
- Can we do better?
- **What problems can you solve?**
- What should you learn from all this?

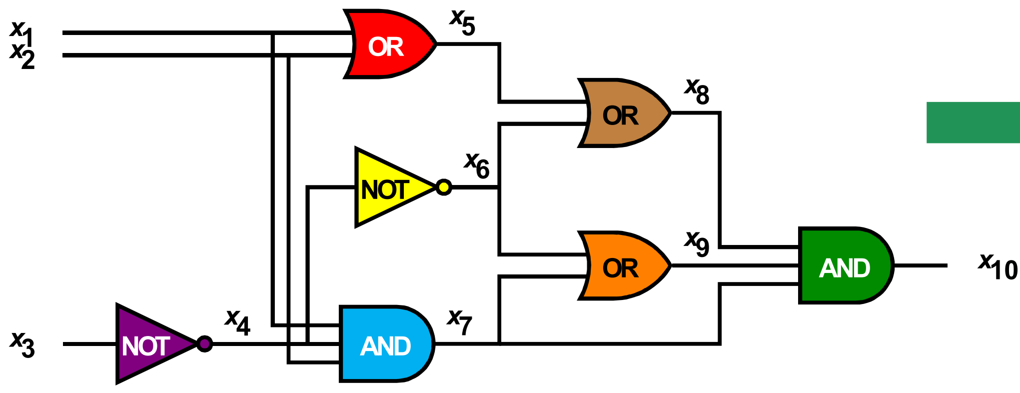
Insight: Easily Solving Inverse Problems

- **Data in an ordinary circuit flows from inputs to outputs**
- **A Hamiltonian has no notion of "inputs" or "outputs", only weighted constraints to satisfy as best as possible**
- **Ergo, a circuit running on a D-Wave system can just as easily run from outputs to inputs**
  - Specify either with $h_i < 0$ for TRUE and $h_i > 0$ for FALSE



  - Nondeterministic in polynomial time (i.e., slow to compute classically)
  - However, solutions to problems in NP can be *verified* in polynomial time (i.e., quickly)
- **Approach to solving problems in NP on a D-Wave**
  - Solve the (easier) inverse problem and run the code *backwards*
- **Caveat**
  - "Solve" doesn't really mean "solve" but rather "heuristically approximate a solution to"

Example 1: Circuit Satisfiability

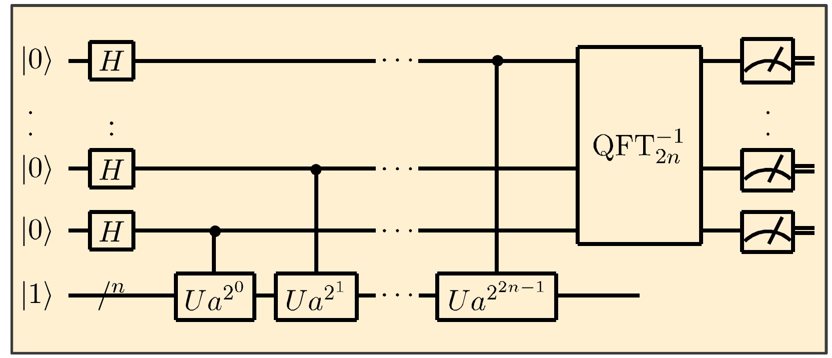


- **Do there exist inputs for which this circuit outputs TRUE?**
- **Classic NP-complete problem—can't beat exhaustive search in the general case (although usable heuristics do exist)**
- **The edif2qmasm approach**
  - Code up the circuit directly and run it backwards from TRUE to a set of inputs

```
module circsat (a, b, c, y);
  input a, b, c;
  output y;
  wire [1:10] x;

  assign x[1] = a;
  assign x[2] = b;
  assign x[3] = c;
  assign x[4] = ~x[3];
  assign x[5] = x[1] | x[2];
  assign x[6] = ~x[4];
  assign x[7] = x[1] & x[2] & x[4];
  assign x[8] = x[5] | x[6];
  assign x[9] = x[6] | x[7];
  assign x[10] = x[8] & x[9] & x[7];
  assign y = x[10];
endmodule
```

Example 2: Factoring

- **NP (but not NP-complete) problem**
- **Even the best known classical algorithms require exponential time**
  - General number field sieve ($O(2^{\sqrt[3]{n}})$)
  - Quadratic sieve ($O(2^{\sqrt{n}})$)
  - Lenstra elliptic curve factorization ($O(2^{\sqrt{n}})$)
  - Many others, all involving lots of tricky number theory
- **A gate-model quantum computer can factor in polynomial time**
  - Shor's algorithm ($O(\log^3 n)$)
  - Involves lots of tricky number theory *and* lots of tricky quantum information processing (e.g., an inverse quantum Fourier transform)
- **The edif2qmasm approach**
  - Express *C* = *A* × *B* in Verilog
  - Run the code backwards from *C* to {*A*, *B*}



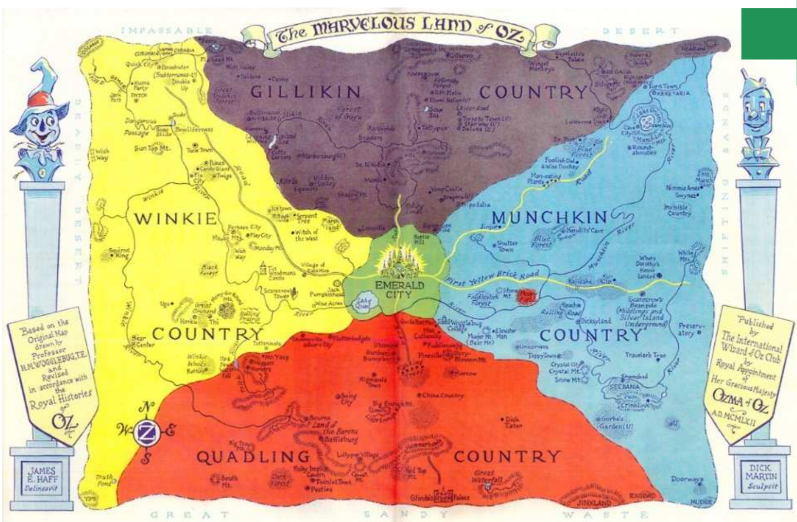
Period-finding component of Shor's algorithm

VS.

```
module mult (multiplicand, multiplier, product);
  input [3:0] multiplicand;
  input [3:0] multiplier;
  output [7:0] product;

  assign product = multiplicand * multiplier;
endmodule
```

Complete Verilog code for factorization

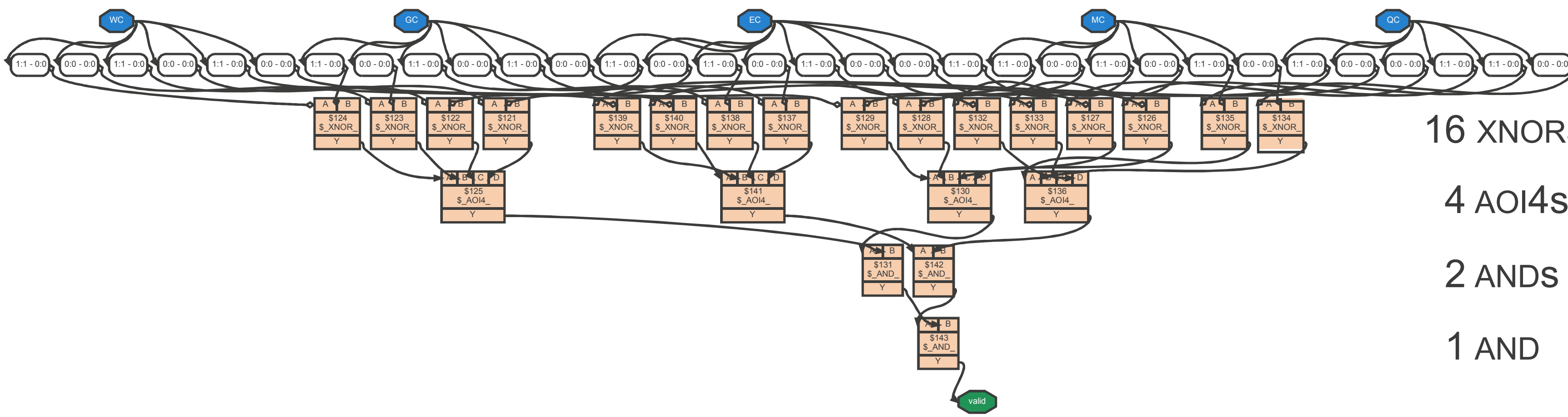
Example 3: Map Coloring




```
module map_color (GC, WC, QC, MC, EC, valid);
  input [1:0] GC;
  input [1:0] WC;
  input [1:0] QC;
  input [1:0] MC;
  input [1:0] EC;
  output valid;
  wire [7:0] tests;

  assign tests[0] = GC != WC;
  assign tests[1] = WC != QC;
  assign tests[2] = QC != MC;
  assign tests[3] = MC != GC;
  assign tests[4] = EC != GC;
  assign tests[5] = EC != WC;
  assign tests[6] = EC != QC;
  assign tests[7] = EC != MC;

  assign valid = &tests[7:0];
endmodule
```

- **Using only four colors, color each region of a planar map such that no two adjacent regions have the same color**
  - NP-complete, with the witness being such a coloring
- **The edif2qmasm approach**
  - Given a coloring, return TRUE if it's valid
  - Run backwards from valid=TRUE to find a valid coloring

Map Coloring after Hardware Synthesis



16 XNORs

4 AOI4s

2 ANDs

1 AND

Map Coloring after Conversion to QMASM

```
!include <stdcell>

!begin_macro map_color
  !use_macro AND $id00014
  !use_macro AND $id00025
  !use_macro AND $id00026
  !use_macro AOI4 $id00008
  !use_macro AOI4 $id00013
  !use_macro AOI4 $id00019
  !use_macro AOI4 $id00024
  !use_macro XNOR $id00004
  !use_macro XNOR $id00005
  !use_macro XNOR $id00006
  !use_macro XNOR $id00007
  !use_macro XNOR $id00009
  !use_macro XNOR $id00010
  !use_macro XNOR $id00011
  !use_macro XNOR $id00012
  !use_macro XNOR $id00015
  !use_macro XNOR $id00016
  !use_macro XNOR $id00017
  !use_macro XNOR $id00018
  !use_macro XNOR $id00020
  !use_macro XNOR $id00021
  !use_macro XNOR $id00022
  !use_macro XNOR $id00023
  EC[0] <-> $id00004.B
  EC[1] <-> $id00005.B
  GC[0] <-> $id00011.A
  GC[1] <-> $id00012.A
  MC[0] <-> $id00016.A
  MC[1] <-> $id00015.A
  QC[0] <-> $id00010.A
  QC[1] <-> $id00009.A
  WC[0] <-> $id00004.A
  WC[1] <-> $id00005.A
  $id00004.A = $id00006.A
  $id00004.A = $id00023.B
  $id00004.B = $id00010.B
  $id00004.B = $id00016.B
  $id00004.B = $id00020.B
  $id00005.A = $id00007.A
  $id00005.A = $id00022.B
  $id00005.B = $id00009.B
  $id00005.B = $id00015.B
  $id00005.B = $id00021.B
  $id00006.A = $id00023.B
  $id00007.A = $id00022.B
  $id00008.A = $id00007.Y
  $id00008.B = $id00006.Y
  $id00008.C = $id00005.Y
  $id00008.D = $id00004.Y
  $id00009.A = $id00018.A
  $id00009.A = $id00022.A
  $id00009.B = $id00015.B
  $id00009.B = $id00021.B
  $id00010.A = $id00017.A
  $id00010.A = $id00023.A
  $id00010.B = $id00016.B
  $id00010.B = $id00020.B
  $id00011.A = $id00006.B
  $id00011.A = $id00020.A
  $id00011.B = $id00017.B
  $id00012.A = $id00007.B
  $id00012.A = $id00021.A
  $id00012.B = $id00018.B
  $id00013.A = $id00012.Y
  $id00013.B = $id00011.Y
  $id00013.C = $id00010.Y
  $id00013.D = $id00009.Y
  $id00014.A = $id00013.Y
  $id00014.B = $id00008.Y
  $id00015.A = $id00012.B
  $id00015.A = $id00018.B
  $id00015.B = $id00021.B
  $id00016.A = $id00011.B
  $id00016.A = $id00017.B
  $id00016.B = $id00020.B
  $id00017.A = $id00023.A
  $id00018.A = $id00022.A
  $id00019.A = $id00018.Y
  $id00019.B = $id00017.Y
  $id00019.C = $id00016.Y
  $id00019.D = $id00015.Y
  $id00020.A = $id00006.B
  $id00021.A = $id00007.B
  $id00024.A = $id00023.Y
  $id00024.B = $id00022.Y
  $id00024.C = $id00021.Y
  $id00024.D = $id00020.Y
  $id00025.A = $id00024.Y
  $id00025.B = $id00019.Y
  $id00026.A = $id00025.Y
  $id00026.B = $id00014.Y
  $id00026.Y = valid
  EC[0] = $id00010.B
  EC[0] = $id00016.B
  EC[0] = $id00020.B
  EC[1] = $id00009.B
  EC[1] = $id00015.B
  EC[1] = $id00021.B
  GC[0] = $id00006.B
  GC[0] = $id00020.A
  GC[1] = $id00007.B
  GC[1] = $id00021.A
  MC[0] = $id00011.B
  MC[0] = $id00017.B
  MC[1] = $id00012.B
  MC[1] = $id00018.B
  QC[0] = $id00017.A
  QC[0] = $id00023.A
  QC[1] = $id00018.A
  QC[1] = $id00022.A
  WC[0] = $id00006.A
  WC[0] = $id00023.B
  WC[1] = $id00007.A
  WC[1] = $id00022.B
!end_macro map_color

!use_macro map_color map_color
```

Map Coloring as a Physical Hamiltonian

- **Not something a human could easily produce**
  - But that's what computers are for
  - And this all came from ~20 lines of easy-to-write, easy-to-read Verilog code

Outline

- How do you program a quantum annealer?
- Can we do better?
- What problems can you solve?
- **What should you learn from all this?**

Conclusions

- **D-Wave systems minimize a *classical* Hamiltonian**
- **…so let's program them with classical programming languages**
  - Argument: Given enough qubits, *any* classical program can be run on a D-Wave
- **Initial choice of language: Verilog**
  - *Pros*: Established language; numerous compilers and development tools (including open-source ones); provides control over bit widths; compiles to simple, easy-to-implement primitives
  - *Cons*: Hardware-centric semantics—may feel odd to Python, C++, Java, … programmers; very limited support for data structures (e.g., arrays and records), floating-point values, and recursion
- **Key benefits of compiling Verilog to a D-Wave Hamiltonian**
  - Easier in most cases to write Verilog code than to prepare a Hamiltonian directly
  - Unlike classical usage, programs can be run backward, from outputs to inputs
- **Insight**
  - Easy but slow: Brute-force solve a computationally expensive problem
  - Difficult but fast: Approximately solve a computationally expensive problem
  - Easy and fast: Use edif2qmasm to approximately solve a computationally expensive problem by solving the simpler inverse problem