Programming a Quantum Annealer

Quantum Computing Seminar
North Carolina State University

Scott Pakin
30 January 2018
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 $\arg \min_{\sigma} H$ with
    $$H = \sum_{i=0}^{N-1} h_i \sigma_i + \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} J_{i,j} \sigma_i \sigma_j$$
  - 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

$$H = \frac{1}{4} \sigma_0 + \frac{1}{4} \sigma_1 - \frac{1}{2} \sigma_2 + \frac{1}{4} \sigma_0 \sigma_1 - \frac{1}{2} \sigma_0 \sigma_2 - \frac{1}{2} \sigma_1 \sigma_2$$
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 $Q_{i,i} = 2(h_i - \sum_{j=0}^{i-1} J_{i,j} - \sum_{j=i+1}^{N-1} J_{i,j})$
  - Example:
    \[ H = \frac{1}{4} \sigma_0 + \frac{1}{4} \sigma_1 - \frac{1}{2} \sigma_2 + \frac{1}{4} \sigma_0 \sigma_1 - \frac{1}{2} \sigma_0 \sigma_2 - \frac{1}{2} \sigma_1 \sigma_2 \quad \leftrightarrow \quad f(x) = x^T \begin{pmatrix} 1 & 1 & -2 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{pmatrix} x \]

  (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 $+1 \equiv$ has the color and $-1 \equiv$ lacks the color

<table>
<thead>
<tr>
<th>$\sigma_{\text{red}}$</th>
<th>$\sigma_{\text{yellow}}$</th>
<th>$\sigma_{\text{green}}$</th>
<th>$\sigma_{\text{blue}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-1$</td>
<td>$-1$</td>
<td>$-1$</td>
<td>$-1$</td>
</tr>
<tr>
<td>$-1$</td>
<td>$-1$</td>
<td>$-1$</td>
<td>$+1$</td>
</tr>
<tr>
<td>$-1$</td>
<td>$-1$</td>
<td>$+1$</td>
<td>$-1$</td>
</tr>
<tr>
<td>$-1$</td>
<td>$-1$</td>
<td>$+1$</td>
<td>$+1$</td>
</tr>
<tr>
<td>$-1$</td>
<td>$+1$</td>
<td>$-1$</td>
<td>$-1$</td>
</tr>
<tr>
<td>$-1$</td>
<td>$+1$</td>
<td>$-1$</td>
<td>$+1$</td>
</tr>
<tr>
<td>$-1$</td>
<td>$+1$</td>
<td>$+1$</td>
<td>$-1$</td>
</tr>
<tr>
<td>$-1$</td>
<td>$+1$</td>
<td>$+1$</td>
<td>$+1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\sigma_{\text{red}}$</th>
<th>$\sigma_{\text{yellow}}$</th>
<th>$\sigma_{\text{green}}$</th>
<th>$\sigma_{\text{blue}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+1$</td>
<td>$-1$</td>
<td>$-1$</td>
<td>$-1$</td>
</tr>
<tr>
<td>$+1$</td>
<td>$-1$</td>
<td>$-1$</td>
<td>$+1$</td>
</tr>
<tr>
<td>$+1$</td>
<td>$-1$</td>
<td>$+1$</td>
<td>$-1$</td>
</tr>
<tr>
<td>$+1$</td>
<td>$-1$</td>
<td>$+1$</td>
<td>$+1$</td>
</tr>
<tr>
<td>$+1$</td>
<td>$+1$</td>
<td>$-1$</td>
<td>$-1$</td>
</tr>
<tr>
<td>$+1$</td>
<td>$+1$</td>
<td>$-1$</td>
<td>$+1$</td>
</tr>
<tr>
<td>$+1$</td>
<td>$+1$</td>
<td>$+1$</td>
<td>$-1$</td>
</tr>
<tr>
<td>$+1$</td>
<td>$+1$</td>
<td>$+1$</td>
<td>$+1$</td>
</tr>
</tbody>
</table>
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

\[ -\mathcal{H}(----) > k \quad -\mathcal{H}(-+++ > k \quad -\mathcal{H}(+-+-) > k \quad -\mathcal{H}(+++-) > k \]
\[ -\mathcal{H}(---+) > k \quad -\mathcal{H}(++++) > k \quad -\mathcal{H}(+-++) > k \quad -\mathcal{H}(+++-) > k \]
\[ -\mathcal{H}(-++++) > k \quad -\mathcal{H}(+++-- > k \quad -\mathcal{H}(++++) > k \]

• 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 \( J_{lj} \) coefficients

– Opposite of what a quantum annealer does

• One possible solution (not unique)

\[ \mathcal{H}(\sigma_r, \sigma_y, \sigma_g, \sigma_b) = \sigma_r + \sigma_y + \sigma_g + \sigma_b + \frac{1}{2} \sigma_r \sigma_y + \frac{1}{2} \sigma_r \sigma_g + \frac{1}{2} \sigma_r \sigma_b + \frac{1}{2} \sigma_y \sigma_g + \frac{1}{2} \sigma_y \sigma_b + \frac{1}{2} \sigma_y \sigma_g \]
\[ \frac{1}{2} \sigma_g \sigma_b \]
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 \( \langle J_{ij} > 0 \) to avoid assigning adjacent regions the same color
  \[
  - \sigma_{w} \sigma_{G} + \sigma_{y} \sigma_{G} + \sigma_{g} \sigma_{G} + \\
  \sigma_{b} \sigma_{G} + \sigma_{e} \sigma_{r} + \sigma_{g} \sigma_{M} + \ldots
  \]

*Oversimplification: OK if neither of two adjacent regions has a given color. Adding \( \mathcal{H} = \sigma_{l} + \sigma_{j} + \sigma_{i} \sigma_{j} \) 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, $\mathcal{H} = \sum_{i=0}^{N-1} h_i \sigma_i + \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} J_{ij} \sigma_i \sigma_j$, 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 $\sigma_i$ for different values of $h_i$:

<table>
<thead>
<tr>
<th>Negative (say, $h_i = -5$)</th>
<th>Zero</th>
<th>Positive (say, $h_i = +5$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma_i$</td>
<td>$h_i \sigma_i$</td>
<td>$\sigma_i$</td>
</tr>
<tr>
<td>-1</td>
<td>+5</td>
<td>-1</td>
</tr>
<tr>
<td>+1</td>
<td>-5</td>
<td>+1</td>
</tr>
</tbody>
</table>

• Observations
  – A negative $h_i$ means, “I want $\sigma_i$ to be TRUE”
  – A zero $h_i$ means, “I don’t care if $\sigma_i$ is TRUE or FALSE”
  – A positive $h_i$ means, “I want $\sigma_i$ to be FALSE”
Interpreting the Problem Hamiltonian (cont.)

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

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

- Here are the optimal values of $\sigma_i$ and $\sigma_j$ for different values of $J_{i,j}$:

<table>
<thead>
<tr>
<th>$\sigma_i$</th>
<th>$\sigma_j$</th>
<th>$J_{i,j} \sigma_i \sigma_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>-5</td>
</tr>
<tr>
<td>-1</td>
<td>+1</td>
<td>+5</td>
</tr>
<tr>
<td>+1</td>
<td>-1</td>
<td>+5</td>
</tr>
<tr>
<td>+1</td>
<td>+1</td>
<td>-5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\sigma_i$</th>
<th>$\sigma_j$</th>
<th>$J_{i,j} \sigma_i \sigma_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>+1</td>
<td>0</td>
</tr>
<tr>
<td>+1</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>+1</td>
<td>+1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\sigma_i$</th>
<th>$\sigma_j$</th>
<th>$J_{i,j} \sigma_i \sigma_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>-5</td>
</tr>
<tr>
<td>-1</td>
<td>+1</td>
<td>-5</td>
</tr>
<tr>
<td>+1</td>
<td>-1</td>
<td>-5</td>
</tr>
<tr>
<td>+1</td>
<td>+1</td>
<td>+5</td>
</tr>
</tbody>
</table>

- Observations
  - A **negative** $J_{i,j}$ means, “I want $\sigma_i$ and $\sigma_j$ to be equal”
  - A **zero** $J_{i,j}$ means, “I don’t care how $\sigma_i$ and $\sigma_j$ are related”
  - A **positive** $J_{i,j}$ means, “I want $\sigma_i$ and $\sigma_j$ to be different”
### Interpretation

- Look what we can express as Hamiltonians so far:

<table>
<thead>
<tr>
<th>Component</th>
<th>Hamiltonian</th>
</tr>
</thead>
<tbody>
<tr>
<td>ground</td>
<td>$H_{\text{GND}} = \sigma_g$</td>
</tr>
<tr>
<td>$V_{\text{CC}}$</td>
<td>$H_{V_{\text{CC}}} = -\sigma_v$</td>
</tr>
<tr>
<td>wire</td>
<td>$H_{\text{wire}} = -\sigma_A \sigma_Y$</td>
</tr>
<tr>
<td>inverter</td>
<td>$H_{\sim} = \sigma_A \sigma_Y$</td>
</tr>
</tbody>
</table>
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 \( Y = A \land B \)**

<table>
<thead>
<tr>
<th>( \sigma_A )</th>
<th>( \sigma_B )</th>
<th>( \sigma_Y )</th>
<th>( \mathcal{H}_\land(\sigma_A, \sigma_B, \sigma_Y) )</th>
<th>Must be</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>(-h_A - h_B - h_Y + J_{A,B} + J_{A,Y} + J_{B,Y})</td>
<td>(= k)</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>+1</td>
<td>(-h_A - h_B + h_Y + J_{A,B} - J_{A,Y} - J_{B,Y})</td>
<td>(&gt; k)</td>
</tr>
<tr>
<td>-1</td>
<td>+1</td>
<td>-1</td>
<td>(-h_A + h_B - h_Y - J_{A,B} + J_{A,Y} - J_{B,Y})</td>
<td>(= k)</td>
</tr>
<tr>
<td>-1</td>
<td>+1</td>
<td>+1</td>
<td>(-h_A + h_B + h_Y - J_{A,B} - J_{A,Y} + J_{B,Y})</td>
<td>(&gt; k)</td>
</tr>
<tr>
<td>+1</td>
<td>-1</td>
<td>-1</td>
<td>(+h_A - h_B - h_Y - J_{A,B} - J_{A,Y} + J_{B,Y})</td>
<td>(= k)</td>
</tr>
<tr>
<td>+1</td>
<td>-1</td>
<td>+1</td>
<td>(+h_A - h_B + h_Y - J_{A,B} + J_{A,Y} - J_{B,Y})</td>
<td>(&gt; k)</td>
</tr>
<tr>
<td>+1</td>
<td>+1</td>
<td>-1</td>
<td>(+h_A + h_B - h_Y + J_{A,B} - J_{A,Y} - J_{B,Y})</td>
<td>(&gt; k)</td>
</tr>
<tr>
<td>+1</td>
<td>+1</td>
<td>+1</td>
<td>(+h_A + h_B + h_Y + J_{A,B} + J_{A,Y} + J_{B,Y})</td>
<td>(= k)</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>$\sigma_A$</th>
<th>$\sigma_B$</th>
<th>$\sigma_Y$</th>
<th>$\sigma_A$</th>
<th>$\sigma_B$</th>
<th>$\sigma_Y$</th>
<th>$\sigma_a$</th>
<th>$\sigma_A$</th>
<th>$\sigma_B$</th>
<th>$\sigma_Y$</th>
<th>$\sigma_a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>+1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
<td>+1</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
</tr>
<tr>
<td>-1</td>
<td>+1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
<td>-1</td>
<td>+1</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
</tr>
<tr>
<td>-1</td>
<td>+1</td>
<td>+1</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
</tr>
<tr>
<td>+1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
<td>-1</td>
<td>+1</td>
</tr>
<tr>
<td>+1</td>
<td>-1</td>
<td>+1</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
</tr>
<tr>
<td>+1</td>
<td>+1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
</tr>
<tr>
<td>+1</td>
<td>+1</td>
<td>+1</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
<td>-1</td>
<td>-1</td>
<td>+1</td>
</tr>
</tbody>
</table>
Increasing our Repertoire

- We can define Hamiltonians for whatever gates we want

<table>
<thead>
<tr>
<th>Gate</th>
<th>Hamiltonian</th>
</tr>
</thead>
<tbody>
<tr>
<td>![AND]</td>
<td>[ H_A = -\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 ]</td>
</tr>
<tr>
<td>![XOR]</td>
<td>[ 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 ]</td>
</tr>
<tr>
<td>![OR]</td>
<td>[ H_V = \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 ]</td>
</tr>
</tbody>
</table>

- Important feature: Hamiltonians can be added
  - Gate + wire + gate = circuit
A Standard Cell Library

• Implement using QMASM, my quantum macro assembler

• 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

\[
\mathcal{H}_A = -\frac{1}{2} \sigma_A - \frac{1}{2} \sigma_B + \sigma_Y + \frac{1}{2} \sigma_A \sigma_B - \sigma_A \sigma_Y - \sigma_B \sigma_Y
\]

# 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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
Hardware Netlists

- **Low-level circuit description**
  - Machine-parseable list of gates and wires

- **Semi-standard: EDIF**
  - Electronic Data Interchange Format

```plaintext
(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 "$_NOT_"
    (cellType GENERIC)
    (view VIEW_NETLIST
      (viewType NETLIST)
      (interface (port A (direction INPUT))
        (port B (direction INPUT))
        (port C (direction INPUT)))
      (contents
        (instance (rename id00004 "$auto$simplemap.cc:37:simplemap_not$49"
          (libraryRef LIB))
          (viewRef VIEW_NETLIST
            (cellRef id00001 (libraryRef LIB))))
          (instance (rename id00005 "$auto$simplemap.cc:85:simplemap_or$50"
            (libraryRef LIB))
            (viewRef VIEW_NETLIST
              (cellRef id00003 (libraryRef LIB))))
            (instance (rename id00006 "$not$example.v:4$2_Y"
              (libraryRef LIB))
              (viewRef VIEW_NETLIST
                (cellRef id00002 (libraryRef LIB))))
                (instance (rename id00007 "$and$example.v:4$1_Y"
                  (libraryRef LIB))
                  (viewRef VIEW_NETLIST
                    (cellRef id00000 (libraryRef LIB))))
                    (design example
                      (cellRef example (libraryRef DESIGN))))))

A -- $48$_AND_ Y
B -- $49$_OR_ Y
C -- $49$_NOT_ Y

30-Jan-2018
Los Alamos National Laboratory
Conversion to QMASM

- **Implement using edif2qmasm**
  - Open-source software, available from [https://github.com/lanl/edif2qmasm](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?

```plaintext
!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
```

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

Los Alamos National Laboratory
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

Diagram:

- Verilog
  - Yosys
  - EDIF
  - edif2qmasm
  - QMASM
  - Physical Hamiltonian
  - Quantum macro assembler
  - Logical Hamiltonian
  - Format-conversion tool
  - Netlist (machine-readable circuit description)
  - Hardware-synthesis tool
  - Hardware-description language
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[6] = ~x[4];
  assign x[9] = x[6] | 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 ($O(2^{3\sqrt{n}})$)
  - 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 $C = A \times B$ in Verilog
  - Run the code backwards from $C$ to $\{A, B\}$

---

**Period-finding component of Shor’s algorithm**

```
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

- 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

```verilog
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
```
Map Coloring after Hardware Synthesis

16 XNORS
4 AOI4s
2 ANDs
1 AND
### Map Coloring after Conversion to QMASM

|---------|----------------------|--------------------------|--------------------------|--------------------------|

**Map Coloring after Conversion to QMASM**

| Use Macro | map_color | map_color | map_color | map_color |

Los Alamos National Laboratory

30-Jan-2018
• 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