Classically Programming a D-Wave Quantum Annealer

D-Wave “Qubits” Users Group Meeting
National Harbor, Maryland

Scott Pakin
26 September 2017
Outline

• Background
• Approach
• Examples
• Conclusions
Background

• A D-Wave system solves exactly one (parameterized) 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

• Thesis: We can map *any* classical problem into this form
  - How? That’s what the rest of this talk is about
  - Why would we want to? We’ll answer that soon enough…
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></th>
<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>
<td>$h_i \sigma_i$</td>
</tr>
<tr>
<td>-1</td>
<td>+5</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>+1</td>
<td>-5</td>
<td>+1</td>
<td>+5</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):

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

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

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

<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>

Zero

<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>

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

<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>$\mathcal{H}_{GND} = \sigma_g$</td>
</tr>
<tr>
<td>$V_{CC}$</td>
<td>$\mathcal{H}_{VCC} = -\sigma_v$</td>
</tr>
<tr>
<td>wire</td>
<td>$\mathcal{H}_{\text{wire}} = -\sigma_A \sigma_Y$</td>
</tr>
<tr>
<td>inverter</td>
<td>$\mathcal{H}_{\neg} = \sigma_A \sigma_Y$</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}_\wedge(\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>
<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>
<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>
<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>
<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>
<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>
<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>
<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>
<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>
<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 gate]</td>
<td>( \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 )</td>
</tr>
<tr>
<td>![XOR gate]</td>
<td>( \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 )</td>
</tr>
<tr>
<td>![OR gate]</td>
<td>( \mathcal{H}_\lor = \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
  - 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

\[ H_A = -\frac{1}{2} \sigma_A - \frac{1}{2} \sigma_B + \sigma_Y + \frac{1}{2} \sigma_A \sigma_B - \sigma_A \sigma_Y - \sigma_B \sigma_Y \]
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")
(git sha1 61f6811, gcc 6.2.0-11ubuntu1 -O2 -fdebug-prefix-map=/build/yosys-0.7sr/yosys-0.7= -fstack-protector-strong -fPIC -Os)
(external LIB
(edifLevel 0)
(technology (numberDefinition))
(cell GND
(cellType GENERIC)
(view VIEW_NETLIST)
(interface (port G (direction OUTPUT))))
(cell VCC
(cellType GENERIC)
(view VIEW_NETLIST)
(interface (port P (direction OUTPUT))))
(cell (rename id00001 "$\_NOT\_"
(cellType GENERIC)
(view VIEW_NETLIST)
(interface
(port A (direction INPUT))
(port B (direction INPUT))
(port C (direction INPUT))))
(library DESIGN
(edifLevel 0)
(technology (numberDefinition))
(cell example
(cellType GENERIC)
(view VIEW_NETLIST)
(interface
(port A (direction INPUT))
(port B (direction INPUT))
(port C (direction INPUT))))
(contents
(instance GND
(viewRef VIEW_NETLIST
(cellRef GND
(libraryRef LIB))))
(instance VCC
(viewRef VIEW_NETLIST
(cellRef VCC
(libraryRef LIB))))
(instance (rename id00004 "$auto$simplemap.cc:85:simplemap_bi$49"
(viewRef VIEW_NETLIST
(cellRef id00001 (libraryRef LIB)))
(interface (port A (direction INPUT))
(port B (direction INPUT))
(port C (direction INPUT))))
(instance (rename id00005 "$auto$simplemap.cc:85:simplemap_bi$50"
(viewRef VIEW_NETLIST
(cellRef id00002 (libraryRef LIB)))
(interface (port A (direction INPUT))
(port B (direction INPUT))
(port C (direction INPUT))))
(instance (rename id00006 "$not$example.v:4$2_Y"
(viewRef VIEW_NETLIST
(cellRef id00003 "$\_OR\_"
(libraryRef LIB)))
(interface (port A (direction INPUT))
(port Y (direction OUTPUT))))
(instance (rename id00007 "$and$example.v:4$1_Y"
(viewRef VIEW_NETLIST
(cellRef id00003 "$\_OR\_"
(libraryRef LIB)))
(interface (port A (direction INPUT))
(port B (direction INPUT))
(port Y (direction OUTPUT))))
(instance (rename id00008 "$or$example.v:4$1_Y"
(viewRef VIEW_NETLIST
(cellRef id00003 "$\_OR\_"
(libraryRef LIB)))
(interface (port A (direction INPUT))
(port B (direction INPUT))
(port Y (direction OUTPUT))))
(net (rename id00009 "$\_AND\_"
(joined (portRef A (instanceRef id00004))
(portRef Y (instanceRef id00004))))
(net Y (joined (portRef A (instanceRef id00006))
(portRef Y (instanceRef id00006))))
(net C (joined (portRef A (instanceRef id00005))
(portRef C))
(net A (joined (portRef A (instanceRef id00005))
(portRef A))
(net B (joined (portRef B (instanceRef id00005))
(portRef B)))))
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
Conversion to QMASM

• Implement using 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]

```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
```
Why Should You Care?

• 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

• Consider problems in the NP complexity class
  – 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
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)
  – NP-hard if you want to know what inputs produce TRUE

• 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 classical algorithms require exponential time
  - General number field sieve \((O(2^{3\sqrt{\pi}}))\)
  - Quadratic sieve \((O(2^{\sqrt{\pi}}))\)
  - Lenstra elliptic curve factorization \((O(2^{\sqrt{\pi}}))\)
  - Many others, all involving lots of tricky number theory
- A 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 \(\{A, B\}\)

\[\begin{array}{c}
0\rangle & H \\
\vdots & \vdots \\
0\rangle & H \\
0\rangle & H \\
1\rangle & Ua^{2^0}Ua^{2^1}\ldots Ua^{2^{2n-1}} \\
\end{array}\]

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-hard to find 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

```
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
# Map Coloring after Conversion to QMASM

```plaintext
!include <stdcell>

!begin_macro map_color

| QC[0]        | $id000010.A | $id00012.B = $id00007.B |
| WC[0]        | $id00004.A  | $id00012.B = $id00018.B |
| AOI4 $id0008 | $id00004.A = $id00006.A |
| AOI4 $id00013| $id00004.B = $id00010.B |
| AOI4 $id00019| $id00004.B = $id00016.B |
| AOI4 $id00024| $id00004.B = $id00020.B |
| XOR $id00004 | $id00005.A = $id00007.A |
| XOR $id00005 | $id00005.A = $id00022.B |
| XOR $id00006 | $id00005.B = $id00009.B |
| XOR $id00007 | $id00005.B = $id00015.B |
| XOR $id00009 | $id00005.B = $id00021.B |
| XOR $id00010 | $id00006.A = $id00023.B |
| XOR $id00011 | $id00007.A = $id00022.B |
| XOR $id00012 | $id00008.A = $id00007.Y |
| XOR $id00015 | $id00008.B = $id00006.Y |
| XOR $id00016 | $id00008.C = $id00005.Y |
| XOR $id00017 | $id00008.D = $id00004.Y |
| XOR $id00018 | $id00009.A = $id00018.A |
| XOR $id00020 | $id00009.A = $id00022.A |
| XOR $id00021 | $id00009.B = $id00015.B |
| XOR $id00022 | $id00009.B = $id00021.B |
| XOR $id00023 | $id00010.A = $id00017.A |
| EC[0]        | $id00004.B  | $id00010.A = $id00023.A |
| GC[0]        | $id00011.A  | $id00010.B = $id00020.B |
| MC[0]        | $id00016.A  | $id00011.A = $id00020.A |

!end_macro map_color

!use_macro map_color

!use_macro map_color

!use_macro map_color

26-Sep-2017
```
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
Conclusions

- D-Wave systems minimize a classical Hamiltonian
- …so let’s program them with classical programming languages
  - Thesis: 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