AQC Domino #1

Modular native logic for adiabatic quantum computation systems
May 24, 2009

http://stuntrabbit.net (draft not yet posted)
[DRAFT] This is rough and goofy. There you have it.

Part 1: Overview and Setup

Purpose
The goal of this series is to develop a set of techniques for building logical structures which can run natively on the Chimera AQC equipment under development at D-Wave Systems. This is exploration. These structures are not intended to be optimally efficient, and are not intended to assert or prove anything scientifically.

Hardware
At the time of writing, D-Wave Systems is beginning functional testing on Chimera, a 128-bit adiabatic quantum computation device. They have provided a simulator for testing programs in a variety of formats. All of the programs described in this paper may be run on the D-Wave simulator or actual AQC hardware. See the **Source Code Listings** at the end of this paper for details.

What follows is a very short and oversimplified description of the Chimera device. For more complete and accurate information, please see the D-Wave website: http//www.dwavesys.com

Essentially the current Chimera chip consists of a 4x4 grid of **cells**. Each cell consists of 8 **nodes** (the qubits). The **links** between nodes are as shown here:

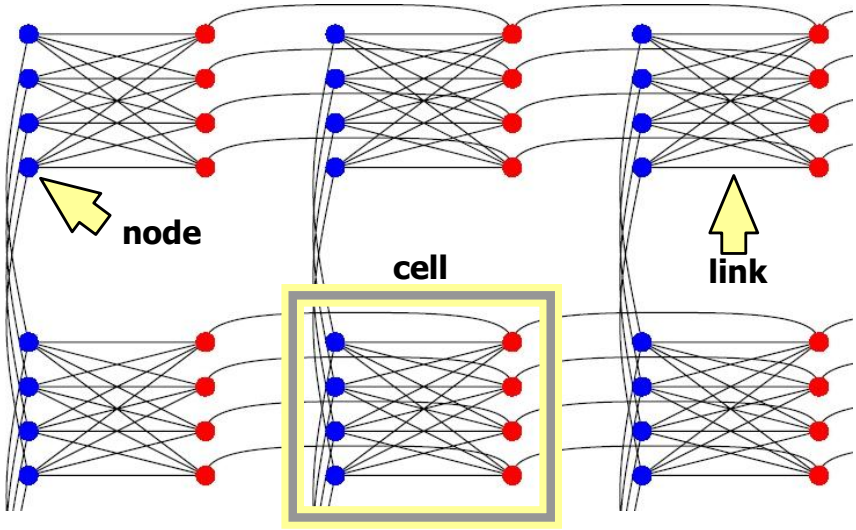


Figure 1.1: Chimera anatomy

Input and Output
- The input data consists of a bias value (any integer) for each node and for each link. For 4x4 cells, using 32-bit integers for bias values, this is a total input data size of **2944 bytes** (128 nodes + 608 links).
- The output data consists simply of one bit per node. For 4x4 cells with 8 nodes per cell, this is a total data size of **16 bytes**.

Evaluation
The **score** of a given output is obtained by adding together the following:
- All node bias values for which the output bit is 1
- All link bias values for which both of the link's node output bits are 1.

Here are some examples:

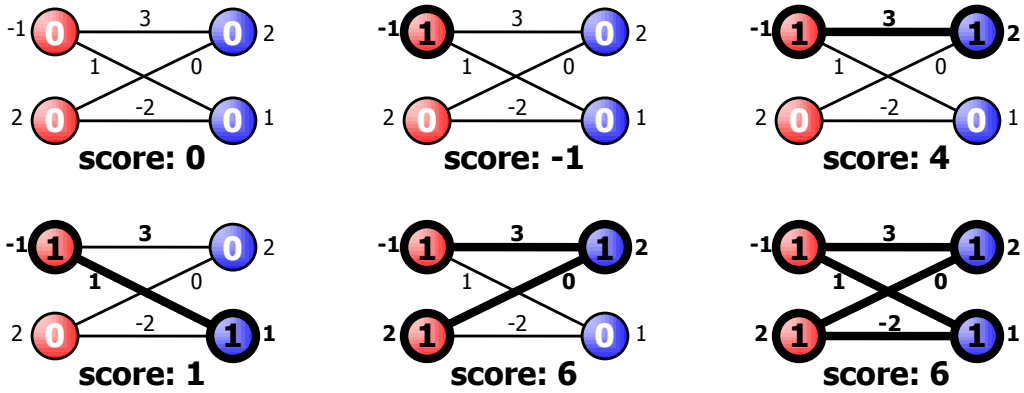


Figure 1.2: Evaluation samples
*(use **Source Code Listing 1.1** to run this program)*

AQC Result
When the AQC is run, it sets the output bits to the configuration which will provide the **minimum score**. If more than one configuration provides the same minimum score, the result is a random selection of the minimum-score options.

This is the whole point of the AQC device. To find the minimum score *without* a quantum device, it is in general necessary to evaluate every possible combination of bits. For the small 4-bit cells in Figure 1.2, that's only 16 combinations to try. You can probably even solve the that example in your head. However, for the 128 bits in the 4x4x8 Chimera, you'll need to try 2^{128} combinations, which would take a fast desktop PC (4GHz quad-core, for example) at least 10^{24} years to complete.

Part 2: Dominoes and Modularity

Cell Visualization
The Chimera documentation diagrams the connectivity of each 8-node cell as follows:

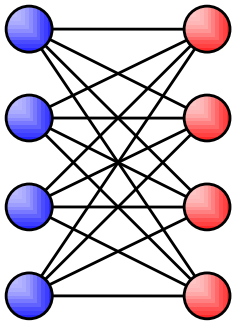


Figure 2.1: One cell

The use of the red and blue colors is purely illustrative. Notice that all blues are connected to all reds, and no two nodes of the same color are connected (within a cell). Also notice that there are no 3-node connected triangles anywhere. This turns our to be an important constraint when it comes to logic design.

Of course, the shape in Figure 2.1 is purely diagrammatic. If it makes things easier, we can reconfigure the nodes in a variety of equivalent and valid ways.

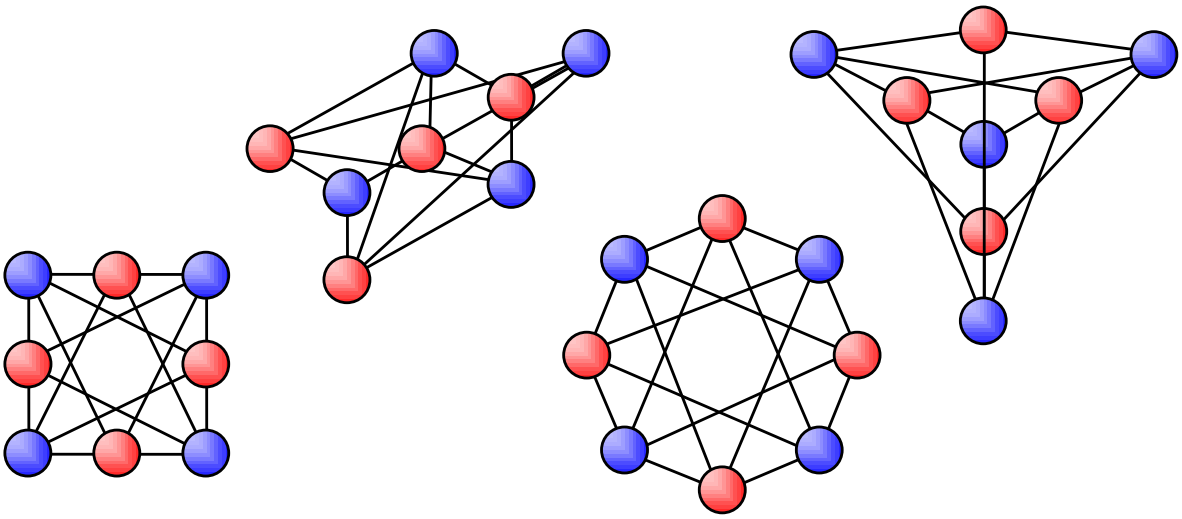


Figure 2.2: A zoo of qubyte variations

While there may be insight to be gained from each of these visualizations, the one we'll focus on in this paper is a 3-dimensional cube, connected along the 12 edges and the 4 major diagonals:

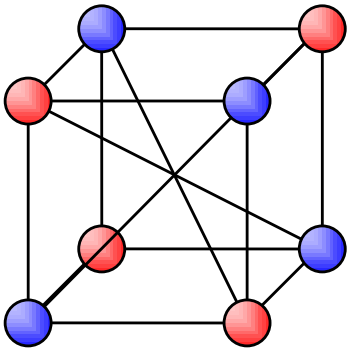


Figure 2.3: Qubit cube

Logic Panels

Now that we've got 8 bits looking smooth and lovely, what we'd like to do is define a useful logical function, and apply it to a single face of the cube. We'll define some useful logic in the next section, but for now let's suppose that a 4-sided **logic panel** like the one in Figure 2.4 does something important:

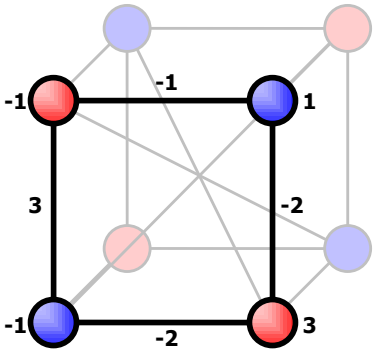


Figure 2.4: One logic panel

Using this cube-shaped model, it's fairly easy to see that we can fit two copies of this remarkably useful whatever-it–is on a single cube allowing each to function independently:

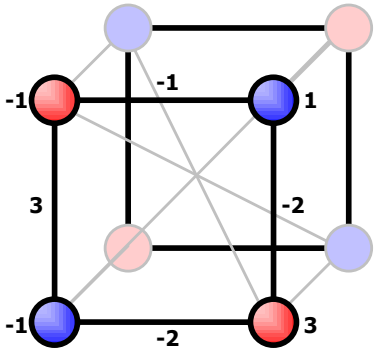


Figure 2.5: Two independent logic panels in the same cell

of course, what we'd really like to do is connect them together in some way, hopefully allowing us to build something larger and potentially more useful.

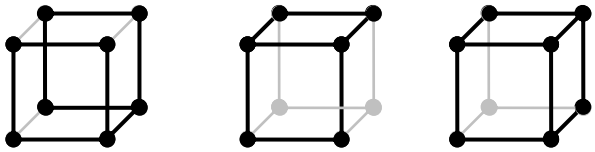


Figure 2.6: A dream of combining logic panels

But now we start to run into some restrictions. When we start to connect two or more panels, they're going to affect or interfere with one another somehow. It turns out that the process of fitting these blocks together resembles a 3-dimensional game of dominoes.

AQC Domino Notation

In light of this, I'm going to make a notation shift here, and switch to a dice/dominoes look. This is not required, but I find it easier to stare at while solving problems. If you find digits easier to use, stick with them; they'll work fine.

The elements of the new notation look like this:

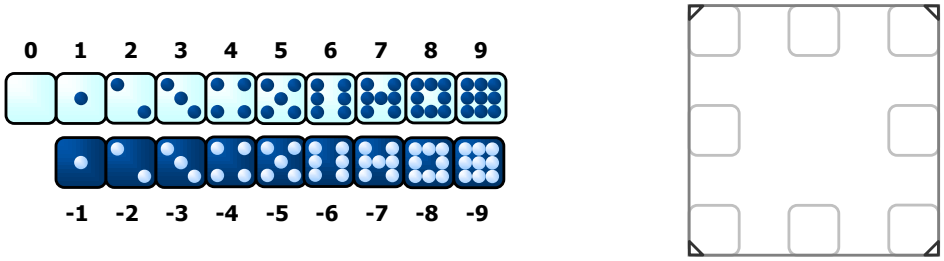


Figure 2.7: Logic panel components

The white dice represent positive numbers, and the black dice represent negative numbers. Translating from the node-based numerical notation to some variations on **AQC Domino** notation looks like this:

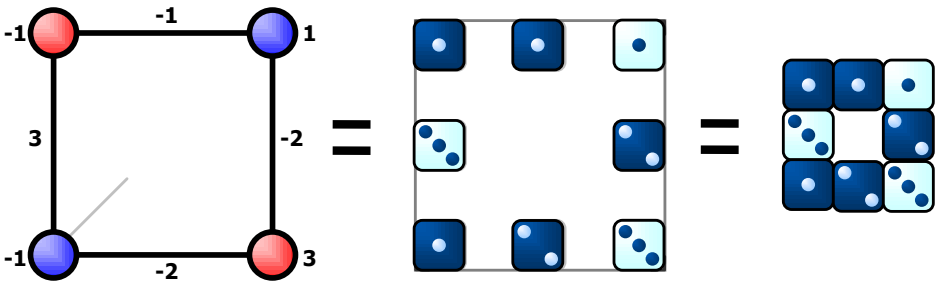


Figure 2.8: Three diagrams of the same logic panel

Having established this notation, we can visually combine available logic panels into functional structures.

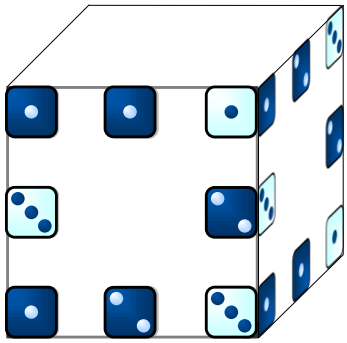


Figure 2.9: But what does it do?

Part 3: One Logic Gate

Score Tables

In the world of digital logic design, truth tables are an important tool for reducing Boolean expressions into something which can be built. We can do something similar with AQC logic panels. Consider the same panel introduced in Part 2, but with the nodes labeled X, A, B and C:

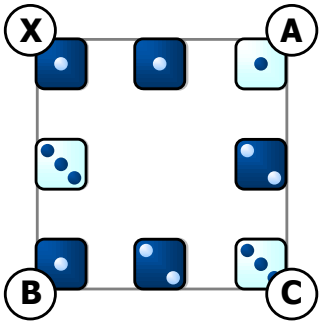


Figure 3.1: Ready for evaluation

If each node will end up with 0 or 1, then there are 16 possible scores, which can easily be expressed in a table:

X	A	B	C	Score
0	0	0	0	0
0	0	0	1	3
0	0	1	0	**-1**
0	0	1	1	0
0	1	0	0	1
0	1	0	1	2
0	1	1	0	0
0	1	1	1	**-1**
1	0	0	0	**-1**
1	0	0	1	2
1	0	1	0	1
1	0	1	1	2
1	1	0	0	**-1**
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

Table 3.1: First evaluation

From this table, we can see that the best (lowest) score is -1, and that there are four different combinations which will provide it. When we run this on the AQC, these are the only results it can return.

Useful Gates

By bringing these "best" rows together, we can see something interesting in A, B and C:

X	A	B	C	Score
0	0	1	0	**-1**
0	1	1	1	**-1**
1	0	0	0	**-1**
1	1	0	0	**-1**

Table 3.2: Distillation

Notice that all four combinations of A/B are represented, and C is 1 only when both A and B are 1. This logic panel is functioning as a logical **AND** gate.

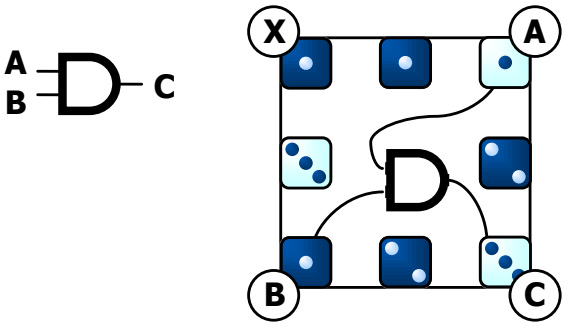


Figure 3.2: And there was logic

The X bit is not part of our gate, but if we remove it, the gate won't work. It's sort of a "scratch" bit. Still, it's worth looking at the table again and noticing that, in this particular case, the X bit happens to be doing something useful as well. It's 1 whenever B is 0, and vice versa. So we have a bonus **NOT** gate in our logic panel.

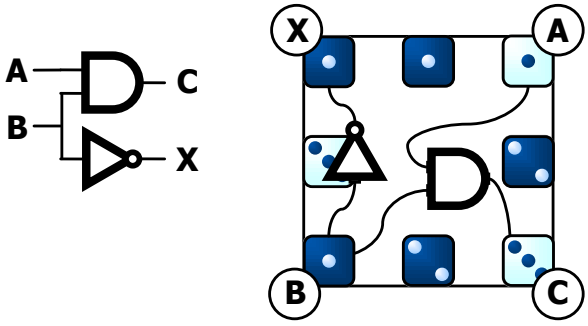


Figure 3.3: Bonus logic
*(use **Source Code Listing 3.1** to run this program)*

Very Strange Reversibility

In conventional digital logic, we're used to gates having "input" and "output", "cause" and "effect". This is something we need to let go of in the world of quantum computation in general. We've *chosen* to interpret Table 3.2 as an AND gate and a NOT gate, but it's equally valid to assume the logic flows the other direction.

For example, the NOT gate between X and B can simply be reversed without changing the table at all. This does not typically work with conventional digital logic; when connected backwards, components tend to catch fire and melt.

Figure 3.4: This "NOT gate" is reversible

We can also reverse other gates, and even express A as a function of X and C, but something very strange happens.

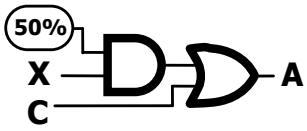


Figure 3.5: Our first example of structural probability

If C is 1, A is always 1. But when X is 1, A has a 50/50 chance of being 1, and none of the other bits (or anything else in the machine) can help us determine which it will be. We'll get much (much) deeper into this in future papers in this series. For now, if you're trying to hand-evaluate a circuit like this, just flip a coin and set it on the 50% marker, and then proceed normally.

Other Single-Panel Gates

With AND and NOT (and the ability to combine them, which we're getting to), we've got everything we need for interesting digital logic. Still, a single-panel NAND solution might be desirable. Fortunately, there are hundreds of combinations which will work. We'll see a good method for finding these in future papers, but here are a few useful panels:

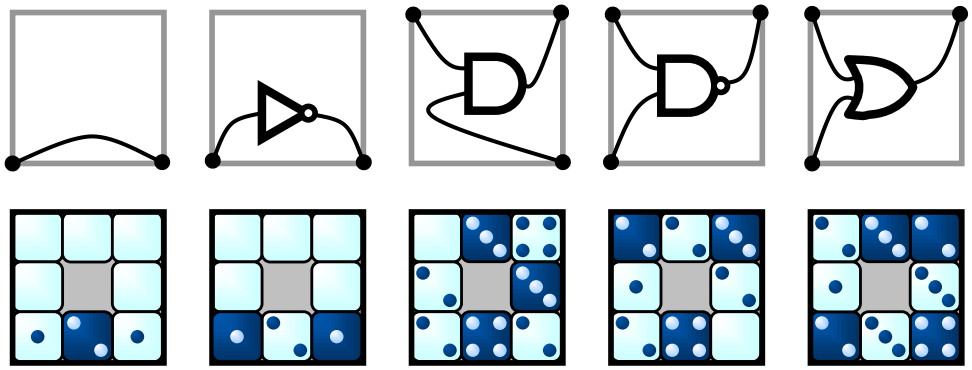


Figure 3.6: A very small rogue's gallery of commonly-used logic panels

With gates like these, we can do things like add, subtract, and multiply. (Because it's all reversible, we might be able to un-multiply as well, which is potentially much more interesting. We'll give this a try before the series is finished.)

Part 4: Gate Combination

Panel Edges

Logic panels with one or two gates are fun, but in order to make them useful, we need to make sure we can combine them without disturbing their functionality.

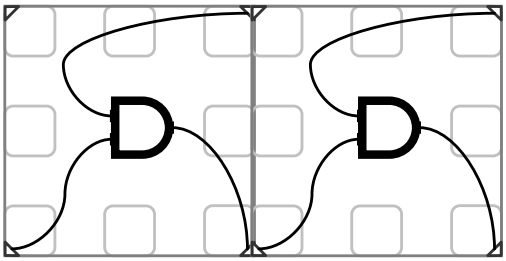


Figure 4.1: Linking panels together

If we find gates with matching edge values and simply place them together, neither of the gates will function at all. In order to connect these panels to one another, we need to add the bias values along the common edge.

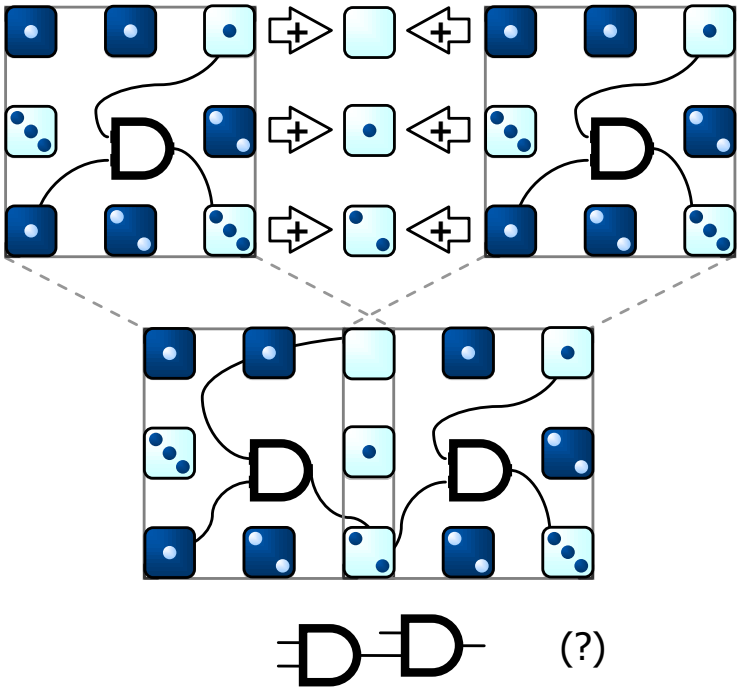


Figure 4.2: Connecting gates by adding the common edge

The good news here is that we don't have to pick matching edges. We can pick any two gates, add them together, and build a logic network.

The bad news is that there's a complication. As seen in the previous section, this panel contains more than a simple AND gate. There's a NOT gate in there as well, and it won't be ignored.

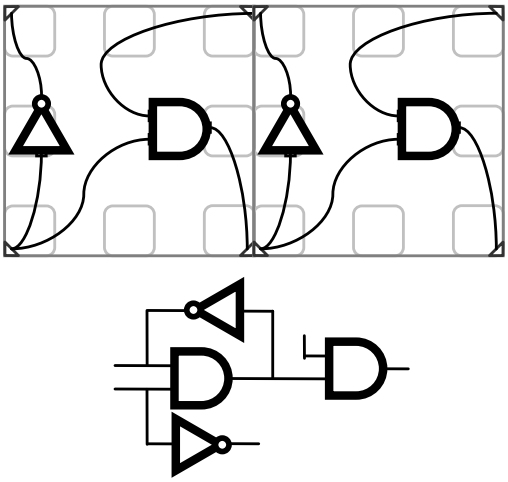


Figure 4.3: A slight complication

In this case, linking these two gates might not produce the logic we intended. Fortunately, there are plenty of other panels to choose from.

Part 5: Putting it Together: An XOR Cube

Constructing a Gate
Later in this series, we'll see that, although there are hundreds of panels for AND, OR, NAND and NOR, there does not seem to be any solution for a 4-node XOR gate. This seems like a great opportunity to try combining gates into an XOR "cube".

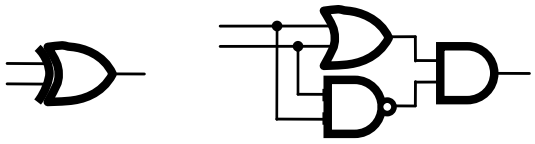


Figure 5.1: XOR can be constructed from other gates

Using an entire cell to construct a single gate is unlikely to be the most efficient use of this state-of-the-art cryogenic processing technology, but it's good practice.

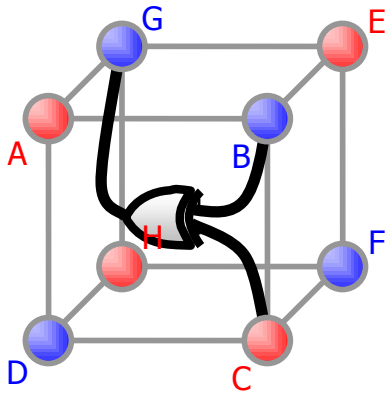


Figure 5.2: XOR Cube

In order to accomplish this, we need to break the XOR gate down into its component gates, map each of those gates onto a face of the cube, and find a way to connect them all together. The following arrangement is one of many possible ways this can be arranged.

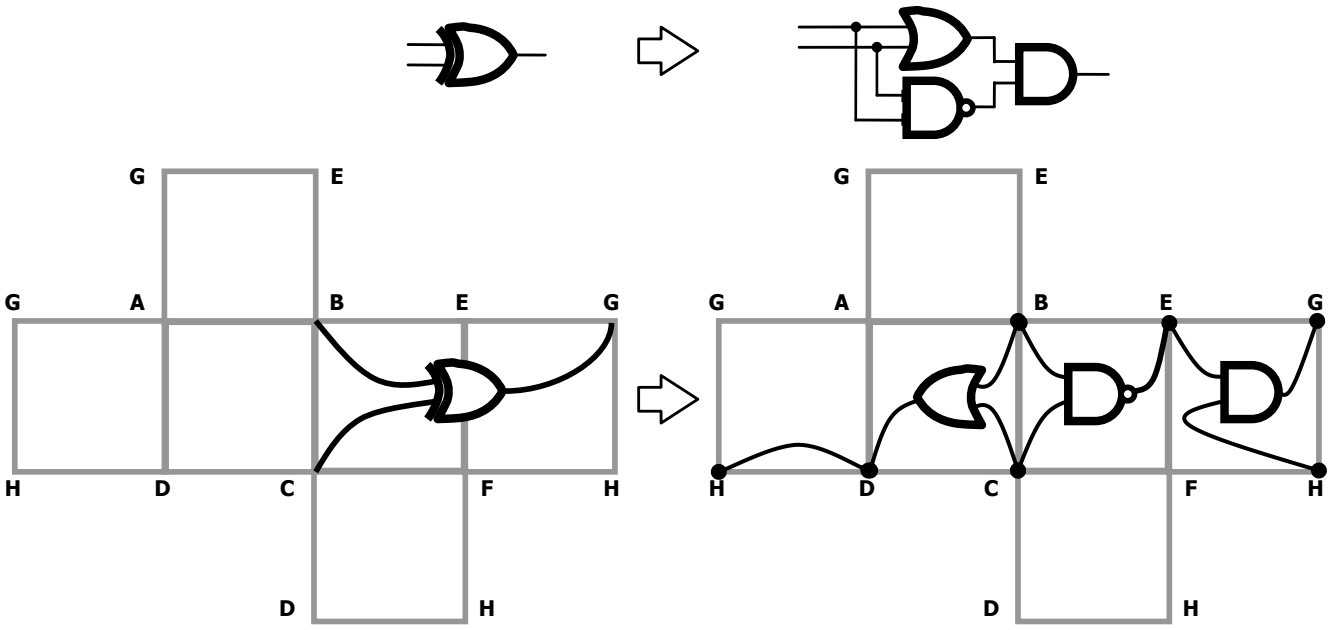


Figure 5.3: Fitting the panels onto a cube

Once we've got an arrangement which seems to fit, we can simply replace the tiles with the ones from Figure 3.6. As mentioned earlier, the process of finding these tiles will be discussed in the next paper in this series.

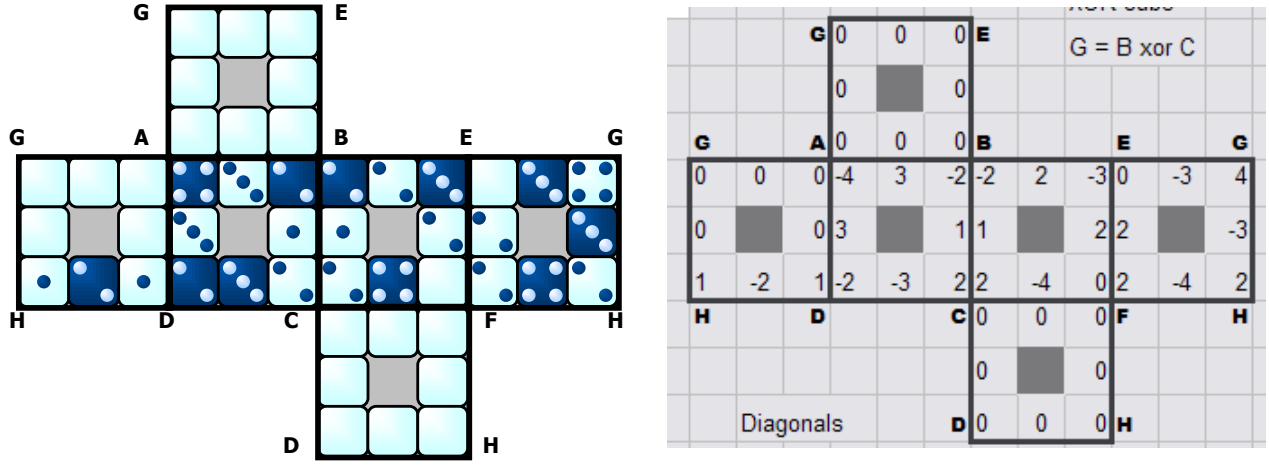


Figure 5.4: Replacing the panel tiles

Now that all of the panels are in place, we need to combine the nodes and edges, as we did in Figure 4.2. Fortunately, this is nothing but addition. A spreadsheet to take care of this automatically can be found at *(need to provide link)*.

Once this edge-reduction has been done, we have the values to feed into the Chimera processor. They look like this:

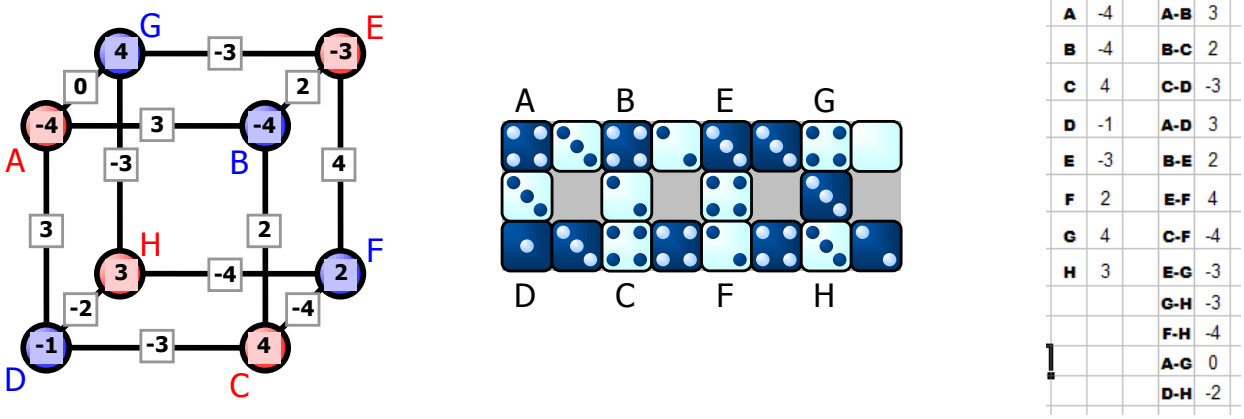


Figure 5.5: The reduced node and link values, ready to be tested

There's one more thing to do, because our nodes are labeled A-H, and the Chimera input system needs a specific node numbering scheme. It turns out that we can just label the red nodes 1-4, and the blue nodes 5-8. Beyond that, the order doesn't matter. It gets slightly more complex later, when we add more cubes.

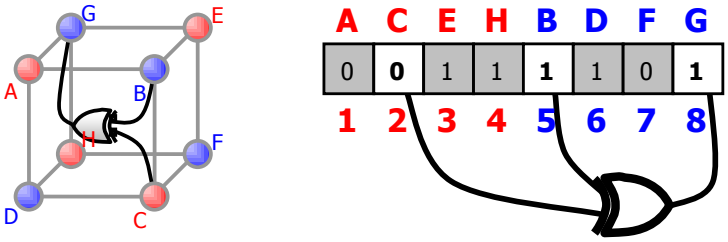


Figure 5.6: Mapping the nodes to Chimera bits
*(use **Source Code Listing 5.1** to run this program)*

This program can be run right away, and you should get the results listed in Source Code Listing 5.1.

Part 6: Node Bits

Setting Input Node Bits
Having functioning gate-level logic is only really useful if you can provide specific bits at the input pins. Fortunately, that's easy. By simply **subtracting 1 from a node's bias value**, you can force the node's bit "on". By adding 1 instead, you can force it "off". This works in all of the Source Code listings in this document.

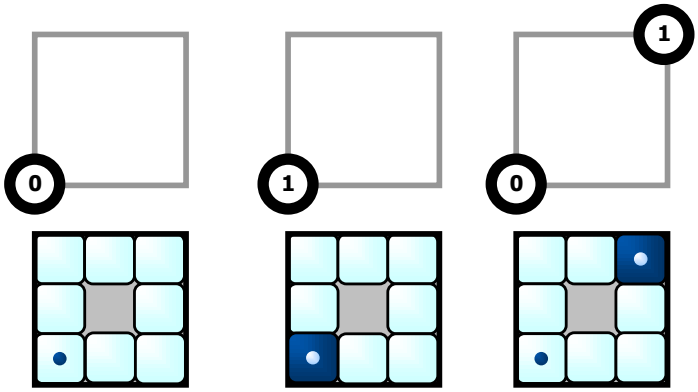


Figure 6.1: Tiles can be used to set individual node values

Setting Output Node Bits
Here's where things get fun; with these logic networks, the concepts of "input" and "output" are just in our heads. By letting go of those, we can just as easily force the output pins to values we want to see on the output, and the input pins will automatically produce valid results.

We can even mix it up, and use a simple addition circuit, *without modifying the logic at all*, to say (for example) "Give me two numbers which add to 11, where one of them is a multiple of 4" and the simple addition circuit will do its best to comply.

Conflicting Constraints
So what if we over-constrain it? Suppose we "force" both sides of a NOT gate to the same value, or hook up some other logically inconsistent setup?

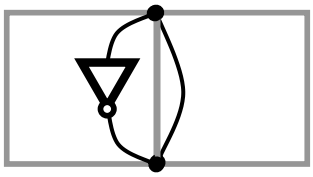


Figure 6.1: This isn't possible to satisfy, and would damage some computers

Fortunately, the universe does not implode (or if it does, that goes unnoticed). You simply get an incorrect result, where one or more constraints is not met. Typically, the answers are easy to double-check.

Overriding Conflicts
If you know there may be conflicts, you can make some gates or constraints "strong" by doubling (or tripling, etc.) the values in their tiles. Their logical functions will still hold, but they'll be able to override "weaker" constraints.

Randomness and Fuzzing
When there's more than one "correct" result (as in the case of a lone AND gate), the actual Chimera chip will return one of the correct results at random. The current Chimera software *simulators*, however, will typically return the same value every time.

To get pseudo-random values, you can simply strengthen your gates (multiply them by, say, 10), and then "fuzz" the nodes, by randomly adding or subtracting 1 from their bias values. The logic should still work, but it will feel more like the actual hardware.

Part 6: What's next

Here's a list of topics I plan to cover in this series. If there's a topic you'd like to see covered, or if you have questions, please contact me at ej@stuntrabbit.net

Constructing Gates and Panels
Hundreds of single-panel gate recipes will be presented, along with the C++ source code for the program I wrote to find them.

A 2x2=4-bit Multiplier
I haven't finished building the multiplier yet. When it's done, it *might* do both multiplication and factoring, just naturally. Whether or not this succeeds, it should make an interesting paper.

Structural Probability
Fun with quantum card tricks and coin flips. Possibly some loaded dice.

Part 7: Acknowledgements

Whether or not this work ever proves genuinely useful, it's an excellent set of logic puzzles to keep me busy. This series would be impossible without the help of two very important people.

My muse, Sue Graham Johnston, is the one who listens to everything I ramble on about, even when it makes no sense at all. She helps me see things I've missed. Using the Domino visual model to help me think things through was her idea in the first place, and the fact that adjacent panels can simply be added was something which I might never have figured out without her questions.

Bill Macready at D-Wave answered every question I sent right away, and always provided extra information I didn't know I was about to need. Bill even asked what I was up to when the questions started getting a little odd.

Appendix 1: Source Code

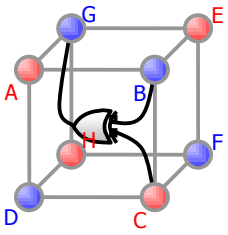
Source Listings: D-Wave Chimera jobs

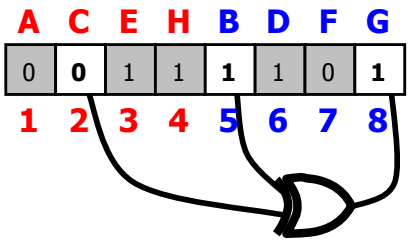
To run these jobs, all you need is a web browser!

- Go to https://apps.dwavesys.com/orionui/pages/webtools/jobAdd
- Choose "Chimera" as **Problem Type**
- Paste the job listing into the **Problem Data** box
- Click Submit

Source Listing 5.1: XOR Cube

This is the XOR cube. Bits 2(C) and 5(B) are the inputs, bit 8(G) is the output.





A	-4	A-B	3
B	-4	B-C	2
C	4	C-D	-3
D	-1	A-D	3
E	-3	B-E	2
F	2	E-F	4
G	4	C-F	-4
H	3	E-G	-3
		C-H	-3
		F-H	-4
		A-G	0
		D-H	-2

Input

This listing contains all of the input data required.

```
1 1 4
1 1 -4
5 5 -4
2 2 4
6 6 -1
3 3 -3
7 7 2
8 8 4
4 4 3
1 5 3
5 2 2
2 6 -3
1 6 3
5 3 2
3 7 4
2 7 -4
3 8 -3
8 4 -3
7 4 -4
1 8 0
6 4 -2
6 3 0
1 7 0
2 8 0
5 4 0
```

Output

If the job succeeds, then it should return a string of bits where bit 8(G) is equal to bit 2(C) **XOR** bit 5(B).