QQFSM(*bf*): A Full-Superposition Quantum Finite State Machine

Updated (January 18, 2015)

Fine Print: This document is not finished, but you're welcome to read it anyway. It has not been peer-reviewed, or even thoroughly researched. These are my own experiments, notes and speculations, with shiny pictures added, and interactive web links. If you have questions, comments, or really good jokes, please contact me at ej@machinelevel.com

Overview

The purpose of this document is to illustrate a step-by-step process for the design of a generic quantum computation device which implements a Turing-equivalent finite state machine capable of using superposition for *all* instructions, data, and state. In other words, it can execute completely separate branching instructions in quantum superposition.

Arguably the single most interesting aspect of quantum computation is the ability to store and operate on many values at once. This device is essentially a simple computer, but one in which the *entire state* is a quantum mechanism which allows simultaneous execution of different programs, which may themselves be quantum-entangled.

Step-by-step development of the device follows, along with fully-functional interactive simulations and notes regarding physical implementation.

The QCEngine Simulator

Throughout this document, I'll use code and examples which run on QCEngine, which is a simulator I've designed and implemented in a few languages. This version allows simple programming of a simulated quantum computer in JavaScript, and has other fun features. Each section will link to examples directly, but the main page is at http://qc.machinelevel.com.

The FSM Base to Implement: (*bf*)

First we need to decide which finite state machine to implement. For simulation as well as physical implementation, the number of qubits used must be kept small. For this reason, I looked for the simplest existing state machine which would suit my needs, and found one called *Brainfuck*, created by Urban Müller, from an article (here) by Brian Raiter. It's a simple language, with eight instructions, which Brian's article conveniently mapped to C:

("**p**" is a pointer, such as char* or int*)

INSTRUCTION	C EQUIVALENT
>	`++p;`
<	`--p;`
+	`++*p;`
-	`--*p;`
.	`putchar(*p);`
,	`*p = getchar();`
[	`while (*p) {`
]	`}`

Using this instruction set, computation can be performed. For example, the following program prints "Hello World!".

```
++++++++++[>+++++++>++++++++++>+++>+<<<<-]>++.>+.+++++++..+++.>++.
<<+++++++++++++++.>.+++.------.--------.>+.>.
```

To test it out on a non-quantum computer, you can try Nayuki's browser-based interpreter (here).

Step 1: Storage Options – QQRAM vs. QQTAPE

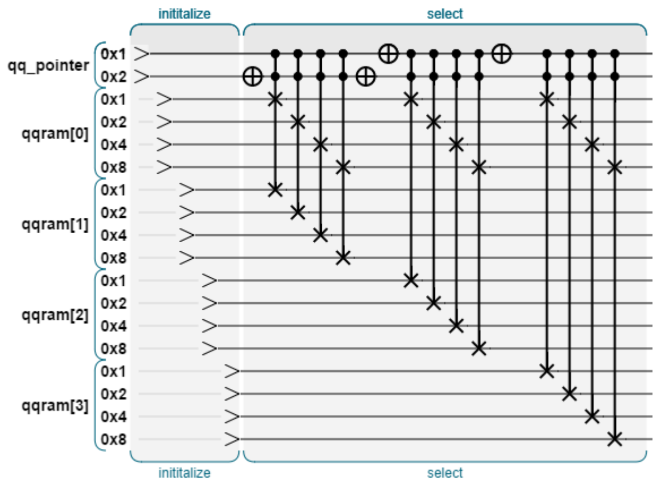
To begin this task, we need a place to store instructions and data. For this, there are two different paradigms to choose between, which I'll call QQRAM and QQTAPE. The "QQ" indicates that it's not just quantum storage, but quantum-addressable as well. This type of storage needs to not only be able to store superposed *values*, but also put them into and get them out of superposed *locations*.

We'll take a look at each option, and make a choice.

Step 1.1: QQRAM (live demo: http://qc.machinelevel.com/qqfsm_examples/qqfsm_01_qqram.html)

QQRAM is a device which uses a qubit-based pointer value, and then performs a "fetch" and "un-fetch" (called "select"), to access the data pointed to. My first implementation of this is fairly straightforward and easy to demonstrate, though not very efficient. The point is to get this machine working, and then we'll optimize it later in this document.

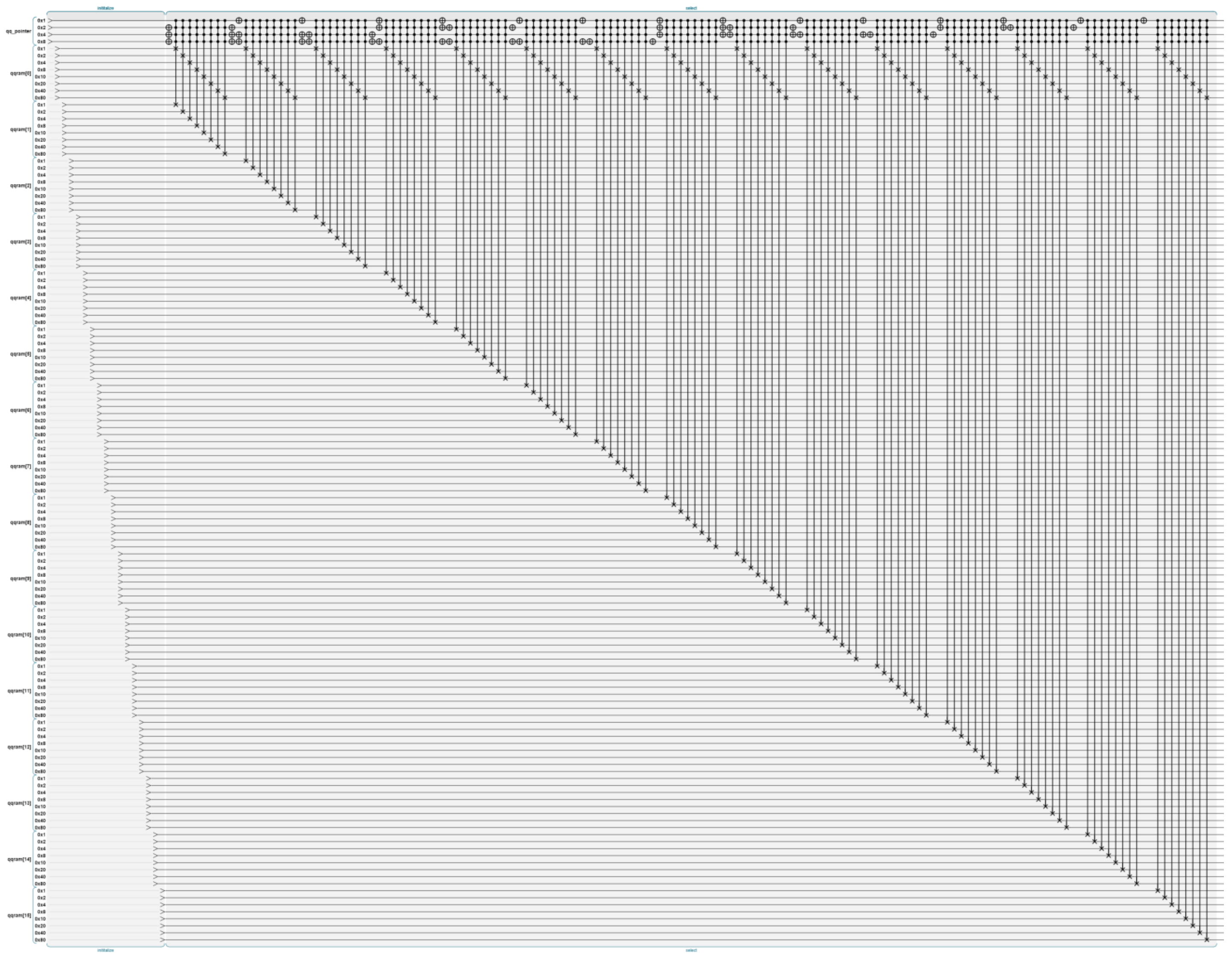
Here's a simple circuit for QQRAM with 4 bits per byte and a 2-bit address (four locations total):



The **initialize** operation simply writes the initial values. After that, we can use the **select** operation to swap location qqram[0] with whatever location is pointed to by qq_pointer. So **select** is essentially a "fetch", and we can think of qqram[0] as the register we're fetching into. The critical feature is that if qq_pointer is a superposition of multiple values, then multiple locations will be fetched at once.

For a live interactive functional demo of this, go here: [link].

This grows very quickly as we increase the address space, obviously. For example, here's a QQRAM implementation for 8-bit bytes and 4-bit addresses (this requires 132 qubits total, and 135 gates):



The actual implementation of this in QCEngine goes like this:

```
var qqram_address_bits = 4;
var qqram_bits_per_byte = 8;

var qqram_address_count = (1 << qqram_address_bits);
var qqram_total_qubits = qqram_address_count * qqram_bits_per_byte;

function qqram_init()
{
  var total_qubits = qqram_total_qubits + qqram_address_bits;
  qc.reset(total_qubits);
  qq_pointer = qint.new(qqram_address_bits, 'qq_pointer');
  qqram = [];
  for (var i = 0; i < qqram_address_count; ++i)
    qqram.push(qint.new(qqram_bits_per_byte, 'qqram[' + i + ']'));
}

function select_qqram(selector)
{
  for (var j = 1; j < qqram_address_count; j++)
    qqram[0].exchange(qqram[j], -1, qintMask([selector, j]), qintMask([selector, ~j]));
}
```

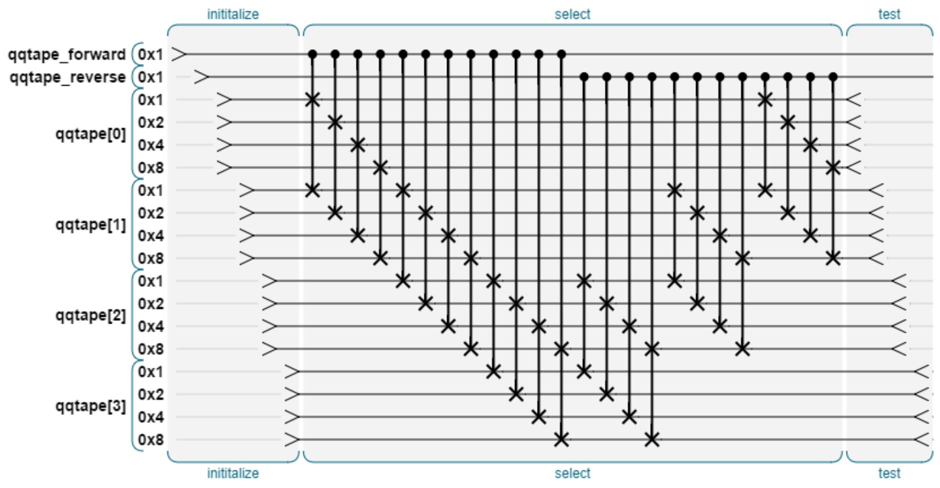
Step 1.2: QQTAPE(live demo: http://qc.machinelevel.com/qqfsm_examples/qqfsm_02_qqtape.html)

Although QQRAM will work to give us qubit-addressable storage, this particular finite state machine may actually not need random access. We might be able to skim away a few qubits by using a simpler mechanism. QQTAPE is an alternate storage mechanism in which we rotate values through memory, effectively stepping our "read head" forward or backward.

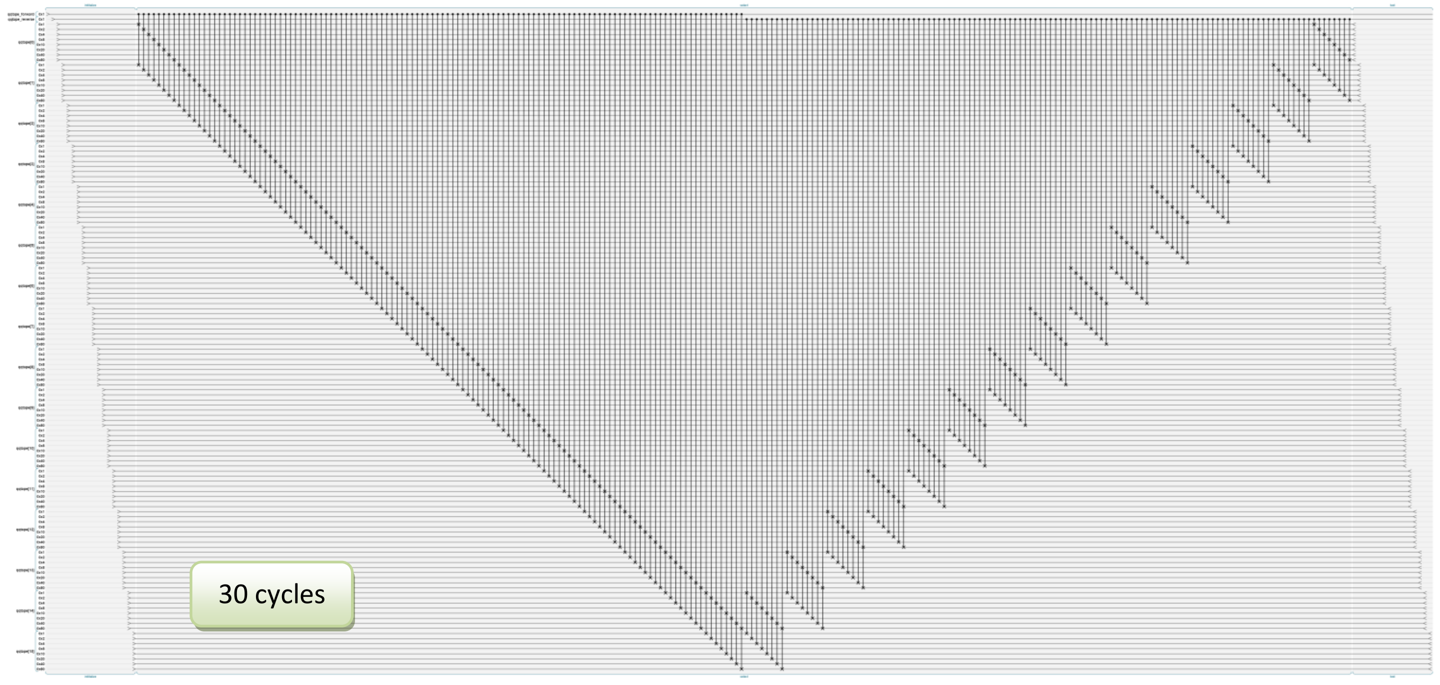
The advantage is that we can still have a completely superposed state, for both positions and values. Also, the tape can be any length; there's no need to stay with powers of two. The disadvantage is that we can't know the absolute address unless we track it separately.

For a live interactive functional demo of this, go here: [link].

Here's a set of gates which implements a QQTAPE system which rolls forward or backward, based on two control qubits. This QQTAPE has 4 bytes of 4 bits each.



Note that since the forward and reverse controls are themselves qubits, this machine is capable of traveling forward *and* reverse *and* staying still, all at once. Here's one which uses 16 bytes of 8 bits each (this requires 132 qubits total, and 240 gates, but it can be done in 30 cycles because each byte's bits are taken in parallel on the hardware.):



While this may look less efficient than the QQRAM version, keep in mind that the QQRAM's **select** operation needs to be performed twice, once to fetch the value and once to put it back.

The actual implementation of this in QCEngine goes like this:

```
var qqtape_bytes = 16;
var qqtape_bits_per_byte = 8;

function qqtape_init()
{
  qqtape_forward = qint.new(1, 'qqtape_forward');
  qqtape_reverse = qint.new(1, 'qqtape_reverse');
  qqtape = [];
  for (var i = 0; i < qqtape_bytes; ++i)
    qqtape.push(qint.new(qqtape_bits_per_byte, 'qqtape[' + i + ']'));
}

function qqtape_roll()
{
  // Roll forward
  for (var i = 0; i < qqtape_bytes - 1; ++i)
    qqtape[i].exchange(qqtape[i + 1], -1, qintMask([qqtape_forward, 1]), 0);

  // Roll backward
  for (var i = qqtape_bytes - 1; i > 0; --i)
    qqtape[i].exchange(qqtape[i - 1], -1, qintMask([qqtape_reverse, 1]), 0);
}
```

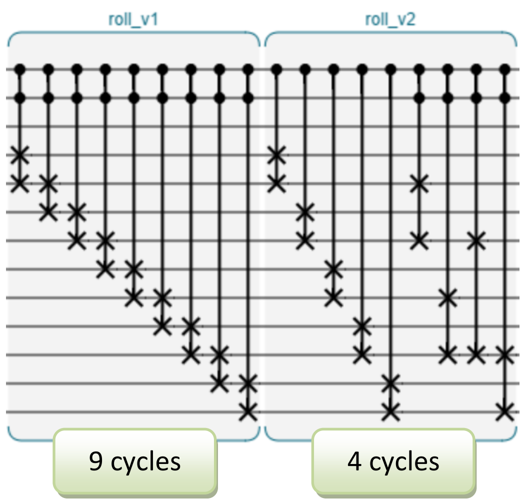
As the *bf* state machine doesn't really require absolute addressing, I'm going to go ahead and implement it using QQTAPE for both instructions and data. (My first shot at this actually used a hybrid, where one storage location was used as QQTAPE for instructions, and QQRAM for data. It worked, but it's a little obfuscated.)

Step 1.3: Significant *(Huge!)* Optimization for QQTAPE

Normally we would wait until the machine is running before optimizing it, but this one is too fun to pass up, so we'll do it now. There are two major issues which can be improved.

1. **Data Dependency**: The "cascade" shape of the operation has every single operation depending on the one before it. Even in non-quantum CPUs, that is a common source of speed problems. If we can re-work this mechanism so that operations can happen in parallel on actual hardware, that will help.
2. **Forward and Reverse**: We're going to need both forward and reverse directions, but if there's a way we can get them to share as many gates as possible, we can reduce our gate count substantially.

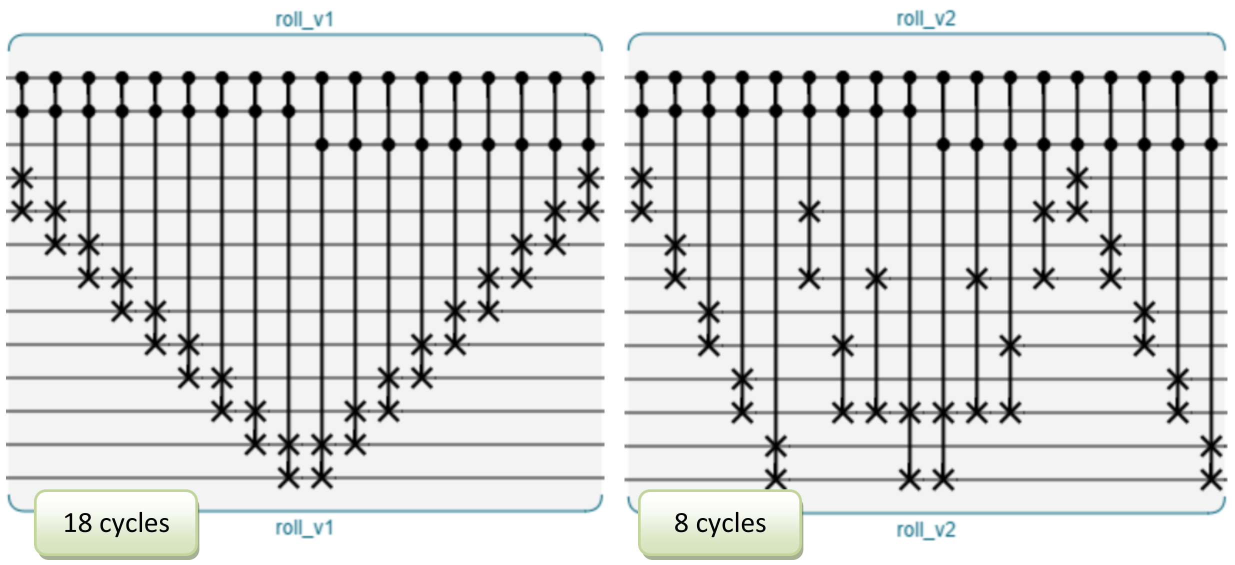
To look at this, we'll pare down to single-bit bytes, just for clarity. First, the **Data Dependency** issue. Considering these two methods for a forward-rolling a 10-qubit QQTAPE…



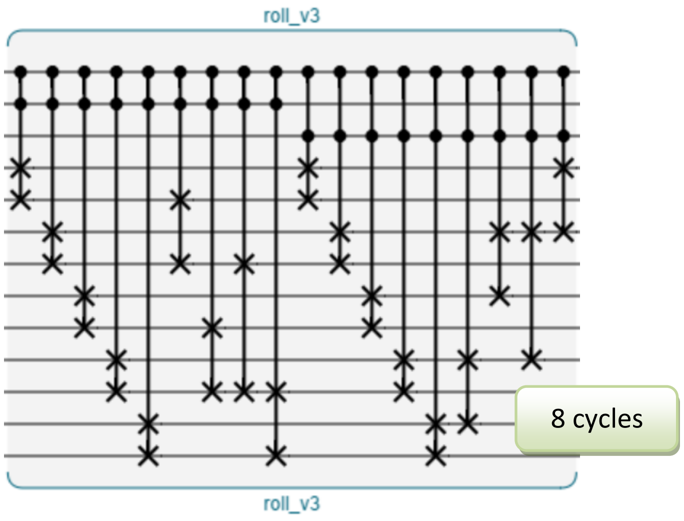
The first method (roll_v1) requires 9 gates, and each must be completed before the next can begin, because their target qubits overlap. So this will take 10 cycles (at least) on actual QC hardware.

The second method (roll_v2) accomplishes exactly the same thing, but in a way which allows most of the operations to happen in parallel. As long as we're okay with overlapping condition bits (more on this later), we can do this operation in 4 cycles. Hooray!

Now, on to the **Forward and Reverse** issue. Assuming we need both, the normal way to reverse a QC operation is just to run it backwards, like this (for v1 and v2 respectively):

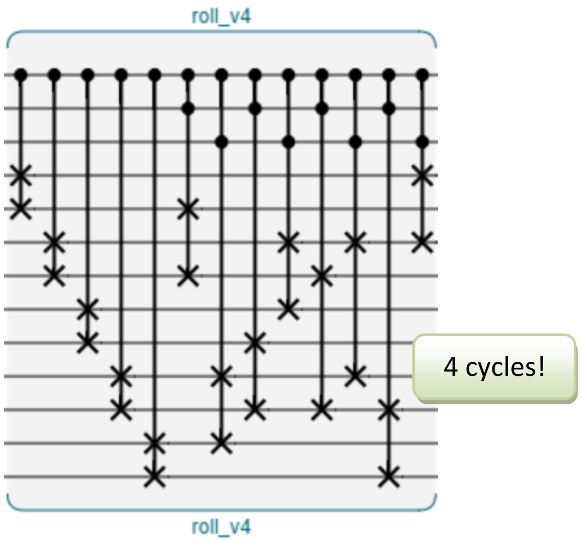


However, our transition to the v2 method brings up a great opportunity. Since our operation is a roll, we can choose to reverse it by flipping it horizontally, but we can get the same effect by flipping it *vertically*. That would look something like this:



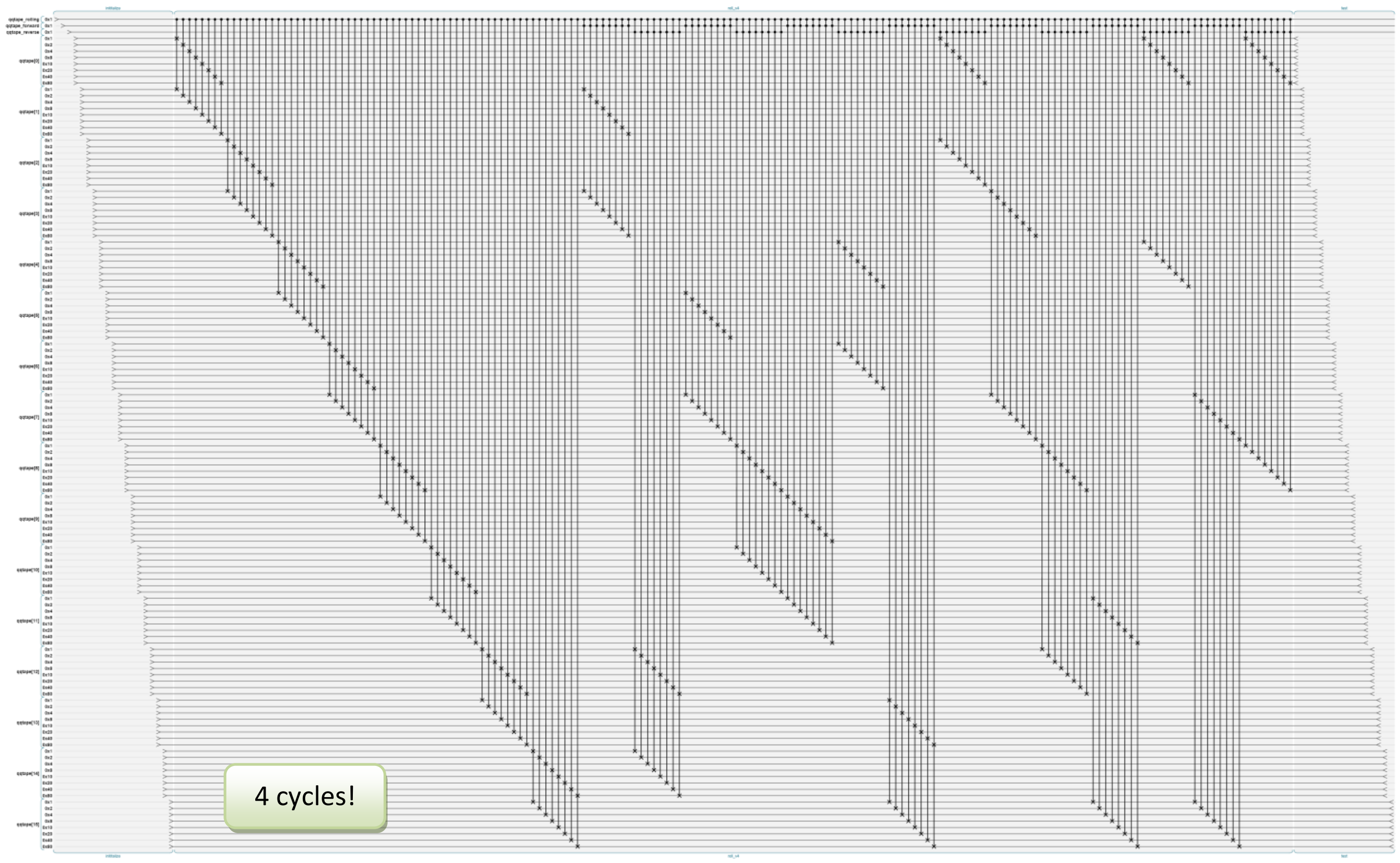
…so why bother with this? Two reasons. First, those first five gates are now identical, and first, so we can simply use a single set of them for both forward and backward. Second, *none* of the remaining gates actually overlap between the forward and reverse directions. So if we interleave them, they can be done in parallel.

So our result handles forward and reverse, and actually parallelizes down to 4 cycles! It looks like this:



There’s one limitation, which is that the QQTAPE length must be an even number. Going from 18 cycles to 4 is a massive speedup, but it gets better as this device grows in complexity. For *n* bytes, the original version requires 2*(n – 1) cycles to execute. Our new version runs in something like log2(*n*) cycles. So for a 100-byte QQTAPE, instead of 198 cycles, we’re looking at about 7. That’s why we did this.

For fun and completeness, here’s the version which has 16 bytes of 8 bits each (like the earlier examples), but should be able to parallelize into about 4 cycles (instead of 32).



…and here's the source for the new roll:

```
// This boils it down to a very efficient forward-and-reverse roller.
function qqtape_roll_v4()
{
  for (var i = 0; i < qqtape_bytes; i += 2)
    qqtape[i].exchange(qqtape[i + 1], -1, qintMask([qqtape_rolling, 1]), 0);

  for (var gap = 2; gap < qqtape_bytes; gap *= 2)
  {
    for (var i = gap - 1; i < qqtape_bytes - 1; i += 2 * gap)
    {
      var j = Math.min(i + gap, qqtape_bytes - 1);
      var ri = qqtape_bytes - i - 1;
      var rj = qqtape_bytes - j - 1;
      qqtape[i].exchange(qqtape[j],   -1, qintMask([qqtape_rolling, 1,
                                                   qqtape_forward, 1]), 0);
      qqtape[ri].exchange(qqtape[rj], -1, qintMask([qqtape_rolling, 1,
                                              qqtape_reverse, 1]), 0);
    }
  }
}
```

Step 1.4: (Clean-up Time) The QQTAPE Class

Since we're going to need multiple QQTAPE objects, we'll wrap them nicely in a class something like this...

```
function QQTape(name, num_bytes, bits_per_byte)
{
  this.name = name;
  this.num_bytes = num_bytes;
  this.bits_per_byte = bits_per_byte;
  this.storage = [];

  this.qubitsRequired = function()
  {
    return num_bytes * bits_per_byte;
  }

  this.activate = function()
  {
    this.storage = [];
    for (var i = 0; i < this.num_bytes; ++i)
      this.storage.push(qint.new(this.bits_per_byte, this.name + '[' + i + ']'));
  }

  this.head = function()
  {
    return this.storage[0];
  }

  this.roll = function(rollCond, rollNot, fwdCond, fwdNot, revCond, revNot)
  {
    for (var i = 0; i < this.num_bytes; i += 2)
      this.storage[i].exchange(this.storage[i + 1], -1, rollCond, rollNot);

    // Roll forward
    for (var gap = 2; gap < this.num_bytes; gap *= 2)
    {
      for (var i = gap - 1; i < this.num_bytes - 1; i += 2 * gap)
      {
        var j = Math.min(i + gap, this.num_bytes - 1);
        var ri = this.num_bytes - i - 1;
        var rj = this.num_bytes - j - 1;
        this.storage[i].exchange(this.storage[j],   -1, fwdCond, fwdNot);
        this.storage[ri].exchange(this.storage[rj], -1, revCond, revNot);
      }
    }
  }
}
```

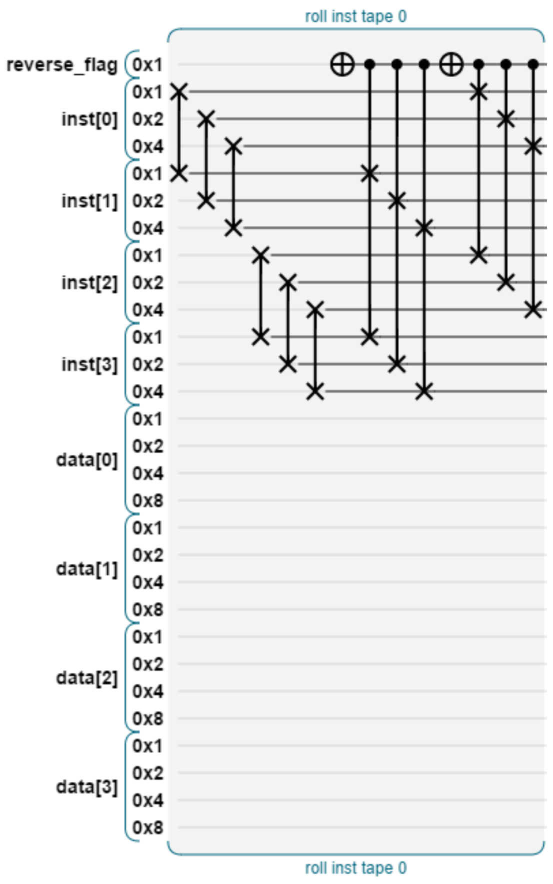
Step 3: The QQFSM(*bf*) Skeleton (live demo: http://qc.machinelevel.com/qqfsm_examples/qqfsm_03_skeleton.html)

Now that we've a storage system we love, we need an actual state machine which can go through the motions, without actually executing instructions. After that, we'll add the instructions.

For this implementation, I'll use separate spaces for instructions and data. I'm not planning to write self-modifying code, but if we want to in the future that's certainly possible.

The skeleton will execute a single complete step of the QQFSM, so at this point it just needs to advance through instruction memory. That's easy.

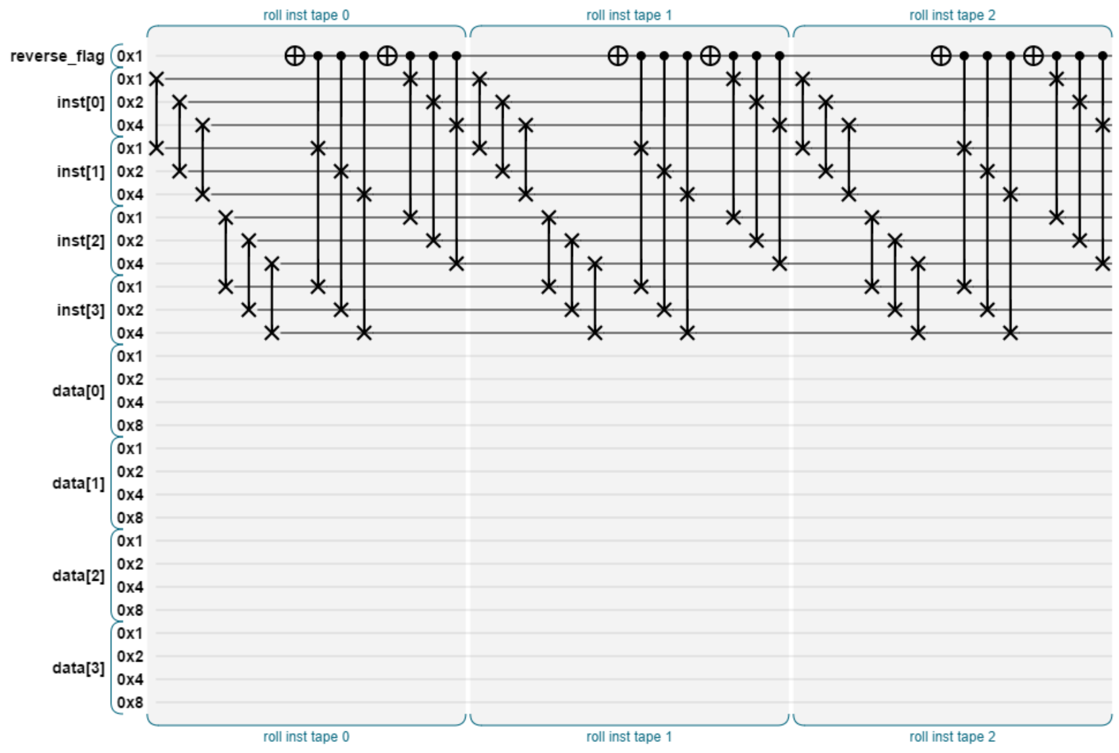
Here's our complete skeleton, with 4x 3-bit instruction slots, 4x 4-bit data slots, and a flag for reverse motion which we don't need yet. At this point, all it does is advance the instruction tape, so the data's never even used. We're about to fix that.



Here's the source for our skeleton so far, with a few small things added that we'll need later…

```
function QQFSM(program, data_bytes, bits_per_byte)
{
  this.program = program;
  this.num_data_bytes = 4;
  this.inst_tape = new QQTape('inst', program.length, 3);
  this.data_tape = new QQTape('data', data_bytes, bits_per_byte);
  this.tick_count = 0;

  this.total_qubits = 1;
  this.total_qubits += this.inst_tape.qubitsRequired();
  this.total_qubits += this.data_tape.qubitsRequired();
  qc.reset(this.total_qubits);
  qc.print('Using ' + this.total_qubits + ' qubits.\n' +
           'RAM est: ' + Math.exp(2, this.total_qubits + 3 - 20) + ' MB.\n');
  this.reverse_flag = qint.new(1, 'reverse_flag');
  this.inst_tape.activate();
  this.data_tape.activate();

  qc.codeLabel('[' + program + ']');
  for (var i = 0; i < this.program.length; ++i)
    this.inst_tape.storage[i].write(translateInstruction(this.program[i]));

  qc.codeLabel('zero mem');
  this.data_tape.setAll(0);

  // Get the instruction values for use as condition bits
  this.inst_incd = translateInstruction('+');
  this.inst_decd = translateInstruction('-');
  this.inst_incp = translateInstruction('>');
  this.inst_decp = translateInstruction('<');
  this.inst_jmp1 = translateInstruction('[');
  this.inst_jmp2 = translateInstruction(']');
  this.inst_in   = translateInstruction('i');
  this.inst_out  = translateInstruction('o');
  // Set up some bits for knowing when to roll the ptr
  this.ptr_roll = this.inst_incp & this.inst_incd & 0x6;
  this.ptr_roll_not = ptr_roll ^ 0x6;

  this.print = function()
  {
    qc.print('------ FSM tick ' + this.tick_count + '\n');
    this.inst_tape.print(instruction_symbols);
    this.data_tape.print(null);
  }

  this.tick = function()
  {
    // Get references to the qubytes at the tape heads
    var inst = this.inst_tape.head();
    var data = this.data_tape.head();

    // Instructions will go here soon!

    qc.codeLabel('roll inst tape ' + this.tick_count);
    var reverse_mask = qintMask([this.reverse_flag, 1]);
    this.inst_tape.roll(0, 0, 0, reverse_mask, reverse_mask, 0);
    this.tick_count++;
  }
}
```

By calling the tick() function multiple times, we can run the machine through its cycle repeatedly…



Step 4: Load Instructions

To load a program into our instruction memory, we really just need a short routine to initialize the instruction qubits, but remember that we also have the option to use superposition. In fact, we can even *entangle* one instruction with another. I haven't thought through all of the ramifications of this yet, but they're interesting for sure.

Now, we're ready to test-run with an actual program! And watch it do almost nothing! Yay.

Suppose we take the program "**+++-**" (yep, kind of a nod to Beethoven's 5th), and run it like this:

```
var qqfsm = new QQFSM('+++-', 2, 4);

qqfsm.print();
for (var t = 0; t < 3; ++t)
{
  qqfsm.tick();
  qqfsm.print();
}
```

Note: Printing the values like that will cause a read() which will of course collapse our quantum state. Still, as a C programmer, printf() is one of only two things in the world that I trust completely. So for now, we'll use it to verify that things are working.

Here's the output:

```
Using 21 qubits.
RAM est: 16 MB.
------ FSM tick 0
  inst: + + + -
  data: 0 0
------ FSM tick 1
  inst: + + - +
  data: 0 0
------ FSM tick 2
  inst: + - + +
  data: 0 0
------ FSM tick 3
  inst: - + + +
  data: 0 0
```

Success! With each tick of the clock, we can see the instruction QQTAPE rolling through its program. The data QQTAPE ought to be changing, but we haven't implemented the **+** and **–** instructions yet. Let's do that right now.

Step 5: New Instructions! (integer math) + –

(live demo: http://qc.machinelevel.com/qqfsm_examples/qqfsm_04_basic_instructions.html)

It’s time to start supporting instructions. Remember, there are eight to do. Some of them are *much* simpler than others, so we’ll start with the simplest.

For the + and – instructions, we just need to increment or decrement the data at the tape head. The gret news is that we’ve set ourselves up for this very nicely. Here we go:

```
data.add( 1, qintMask([inst,  this.inst_incd]),
             qintMask([inst, ~this.inst_incd]));
```

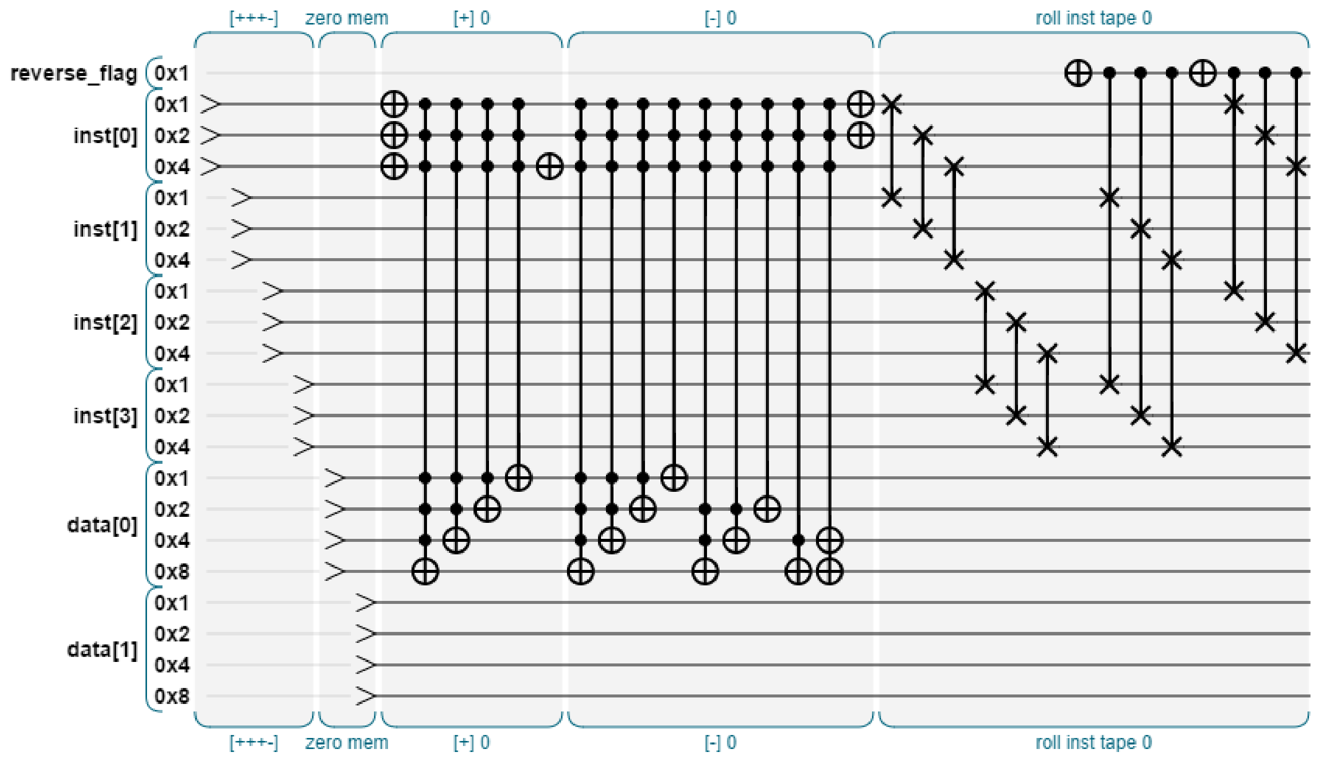
For adding, that’s really all there is to it. We add 1 to the value, with condition bits set to identify the instruction. To subtract, we’ll do this.

```
data.add(-1, qintMask([inst,  this.inst_decd]),
             qintMask([inst, ~this.inst_decd]));
```

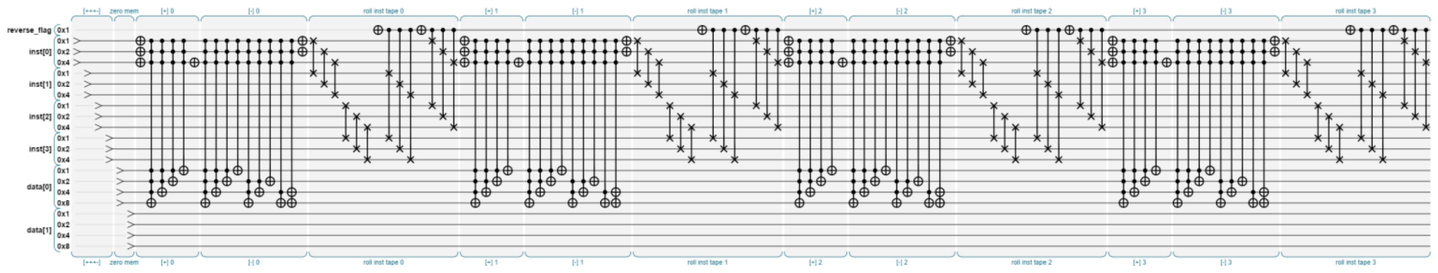
…and then we can just run our test again, and look at the output:

```
Using 21 qubits.
RAM est: 16 MB.
------ FSM tick 0
  inst: + + + -
  data: 0 0
------ FSM tick 1
  inst: + + - +
  data: 1 0
------ FSM tick 2
  inst: + - + +
  data: 2 0
------ FSM tick 3
  inst: - + + +
  data: 3 0
------ FSM tick 4
  inst: + + + -
  data: 2 0
```

Hot diggity! The data value increases every time we hit a + instruction, and then decreases when we hit that final -. What we have here is a calculator which can add up numbers using quantum superposition, as long as we stop looking at the answers. Here’s what one cycle of the the logic looks like:

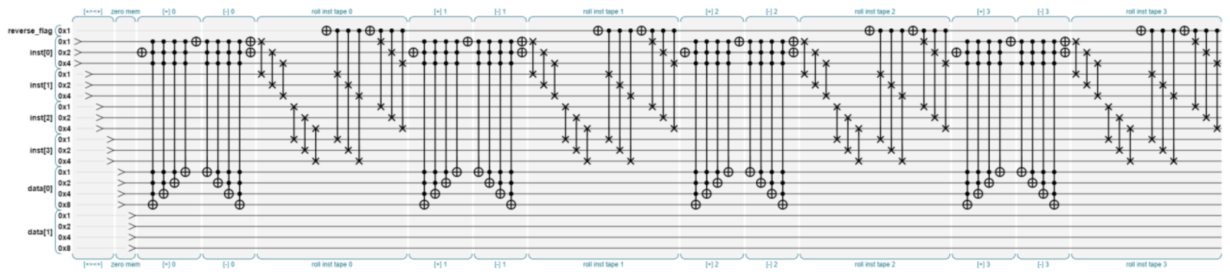


…and then the four cycles we ran look like this:



Hmmmm: While writing the upcoming section on Reversibility, I've just realized that I'm being silly and wasting gates with my `add(-1)`. It really shouldn't need to be any more complex than running the `add(1)` backwards. Fixing that will make the entire engine substantially faster.

Sure enough, fixing that in the engine makes everything smaller and faster, making each subtraction (or addition of a negative constant) twice as fast as it used to be. Looks better too. Here's the exact same thing after the fix:



Step 6: New Instructions! (pointer math) < >

(live demo: http://qc.machinelevel.com/qqfsm_examples/qqfsm_04_basic_instructions.html)

That was easy and fun, so now let's add the data pointer increment and decrement. To do that, we're going to roll data memory, just like we've been rolling instruction memory, except that it's not *always* going to move.

Still, it's pretty easy. Here's all of the code:

```
    this.data_tape.roll(qintMask([inst, this.ptr_roll]),
                        qintMask([inst, this.ptr_roll_not]),
                        qintMask([inst, this.inst_incp]),
                        qintMask([inst, ~this.inst_incp]),
                        qintMask([inst, this.inst_decp]),
                        qintMask([inst, ~this.inst_decp]));
```

So what's going on here? The "roll" function we set up for our QQTAPE takes six arguments, which are three pairs of qubit masks. We use the first two (rollCond, rollNot) to tell QQRAM if *any* rolling is happening. It just uses those as the condition and not-condition for the gates. The next two are conditions to activate the forward-direction rollers, and the last two activate the reverse-direction rollers.

The simple machine we've been testing has only two bytes of data memory, so backward and forward are the same roll. Let's up it to three, and test with a new program:

```
      var qqfsm = new QQFSM('+><+', 4, 4);
```

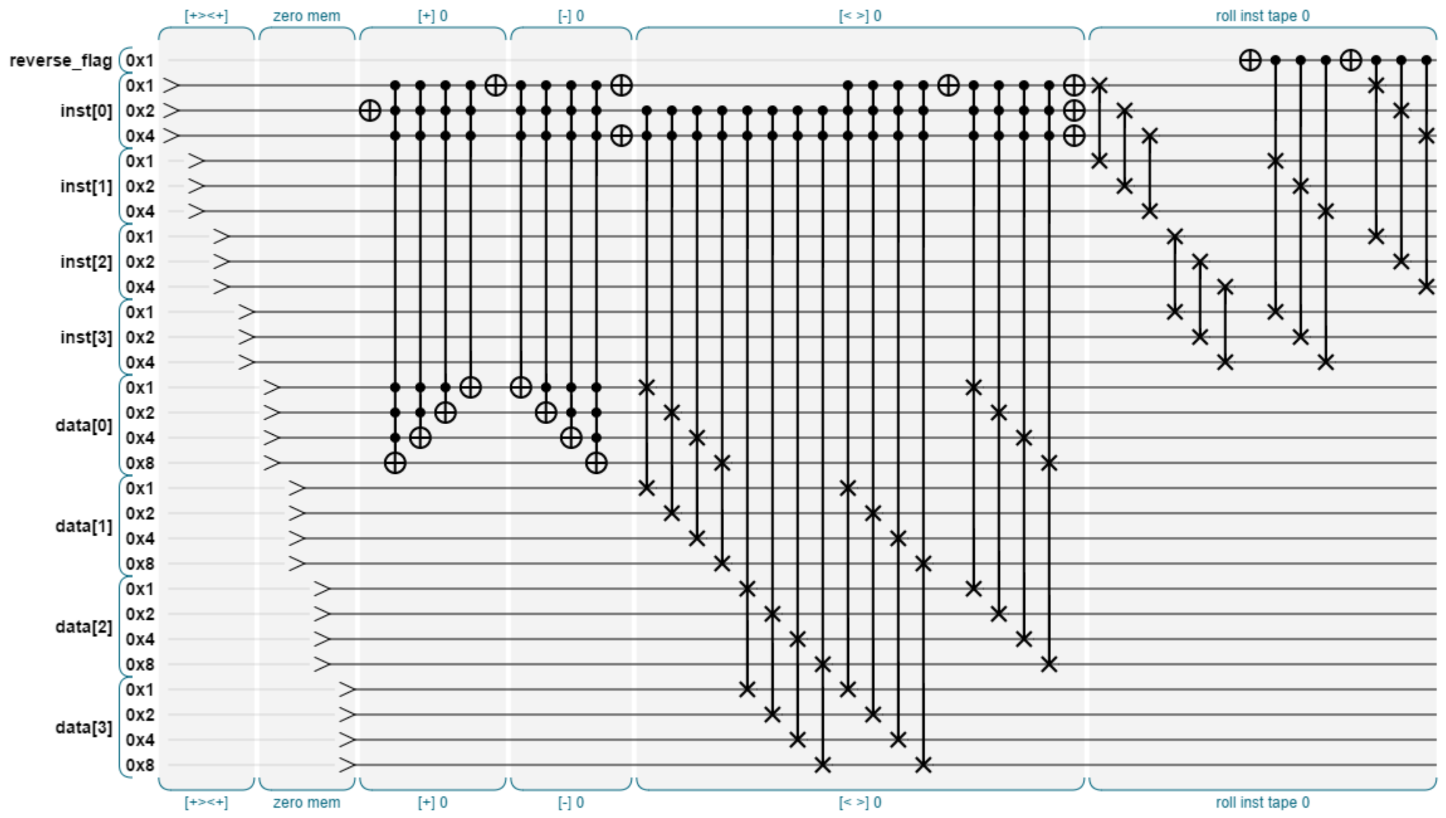
…and running it for 4 ticks, we get this output:

```
      Using 29 qubits.
      RAM est: 4096 MB.
      ------ FSM tick 0
        inst: + > < +
        data: 0 0 0 0
      ------ FSM tick 1
        inst: > < + +
        data: 1 0 0 0
      ------ FSM tick 2
        inst: < + + >
        data: 0 0 0 1
      ------ FSM tick 3
        inst: + + > <
        data: 1 0 0 0
      ------ FSM tick 4
        inst: + > < +
        data: 2 0 0 0
```

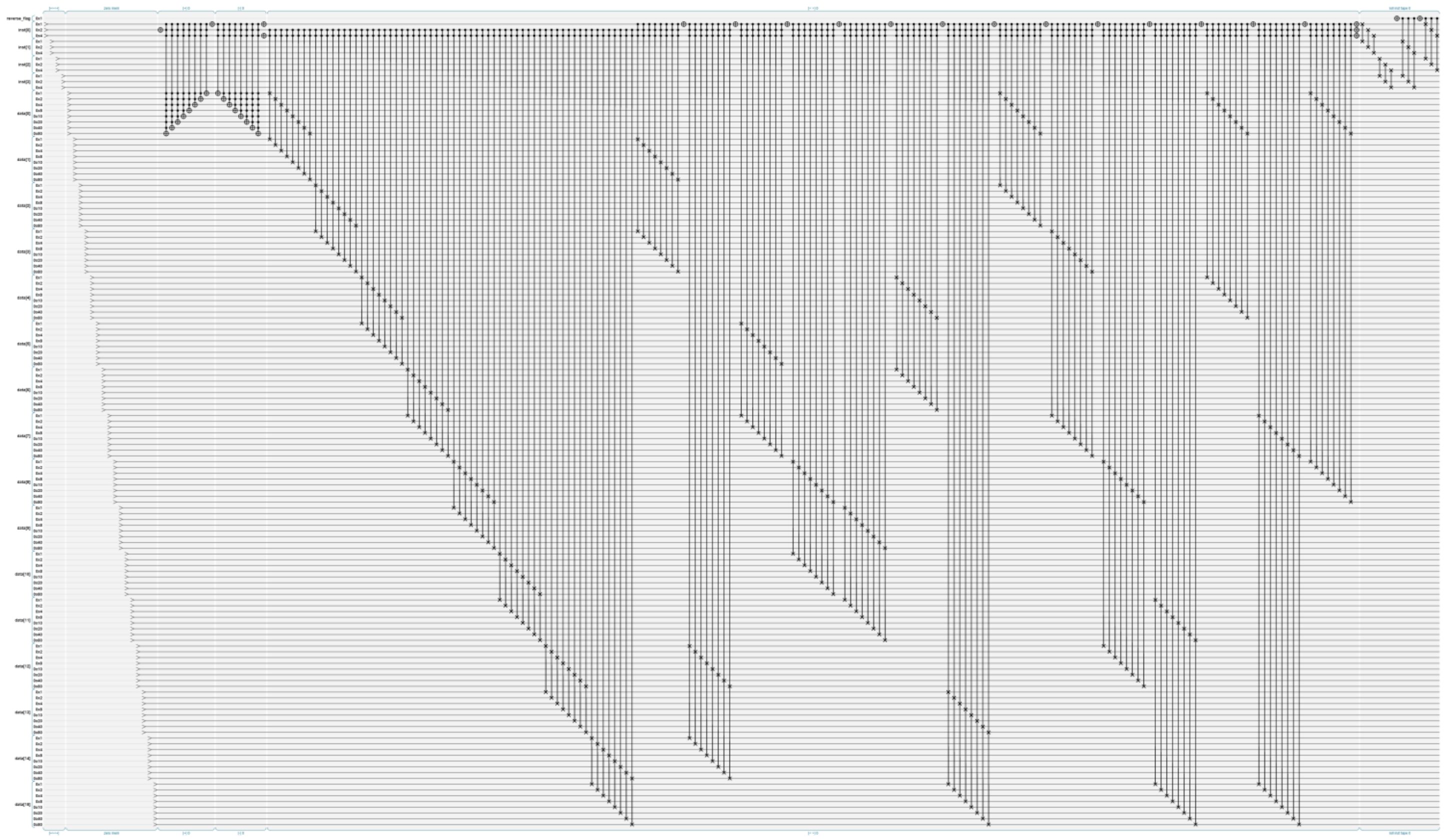
Success again! Our program succeeded in incrementing a data value, moving the pointer, moving it back, and incrementing the value again.

Note: The output mentions that we've grown to 29 qubits, which will take about 4 GB of RAM to simulate. Fortunately, if our computer's not up to that, the QCEngine will fall back to a digital-only sim. Our programs will still run, but all quantum superposition effects won't work.

So now our QQFSM*(bf)* can do everything except branching and I/O, with full quantum superposition goodness. Here's what it looks like so far (using 4 bytes of 4 bits each, just looking at one tick):



Just for fun, the 16 bytes 8-bit-per-byte version requires 141 qubits, and looks like this:



It’s a spiffy-looking gizmo, and it should be pretty fast on actual hardware if our gates parallelize like we think they should.

Step 7: New Instructions! (branching) [ ]

(live demo: http://qc.machinelevel.com/qqfsm_examples/qqfsm_05_branching.html)

At this point we've done all of the easy and fun parts, and it's time to face some serious issues, like **reversibility** and **data loss**. The branching instructions are the part which took me the longest (by far) to sort out.

On Reversibility – After the initialization, every part of the FSM so far is completely reversible. That is, if you run a program (any number of ticks), then run exactly program backwards, you get back where you started. No information is ever lost, so the whole machine is reversible.

Now consider this *bf* program: `[-]`

Looking up the symbols for their C equivalents, we can translate it to: `while (*p) { --*p; }`

…so the program will enter this loop with some value, and then loop until it's zero, and then continue. How can that be reversible? When you run it backwards, p starts at zero, but there's no indication of how many times the loop should be run.

This branching mechanism **loses information**. Throwing away information takes energy. It's strange, because in classical digital logic, the most basic gates which everything else is built on (and/or gates) lose information every time they're used. Two inputs go in, and what comes out (A and B) does not contain enough information to reverse the operation.

While I was trying to construct the branch gates, everything I tried using normal quantum logic *almost* worked, but not quite. Once I realized I had to account for waste, it became easy but expensive.

So we basically have two options:

1. **Clobber** - Each tick, clear one bit to zero. This process us ugly, and interrupts the flow of our program. Even worse, it may collapse the quantum superposition, because information is leaving he system and becoming visible to observers.
2. **Stash** - Allocate enough extra qubits to store one fresh bit for every single possible branch. Hide the extra bits behind your back and say "nope, no waste here." It's sort of like building a "pollution free" car by attaching a very large bag to the tailpipe.

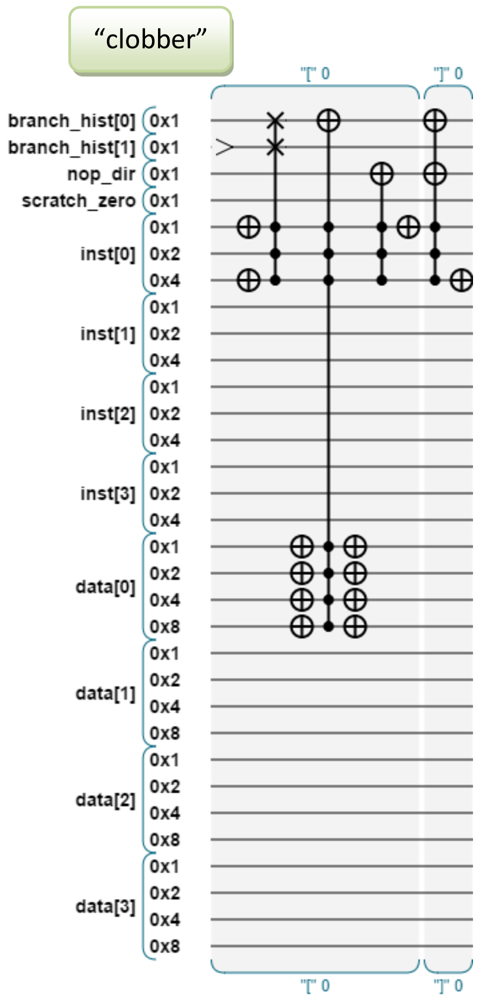
Adding qubits, especially a lot of them, is bad news for both simulations and physical QC's. You could store them up for a while and then clear them all at once, I suppose.

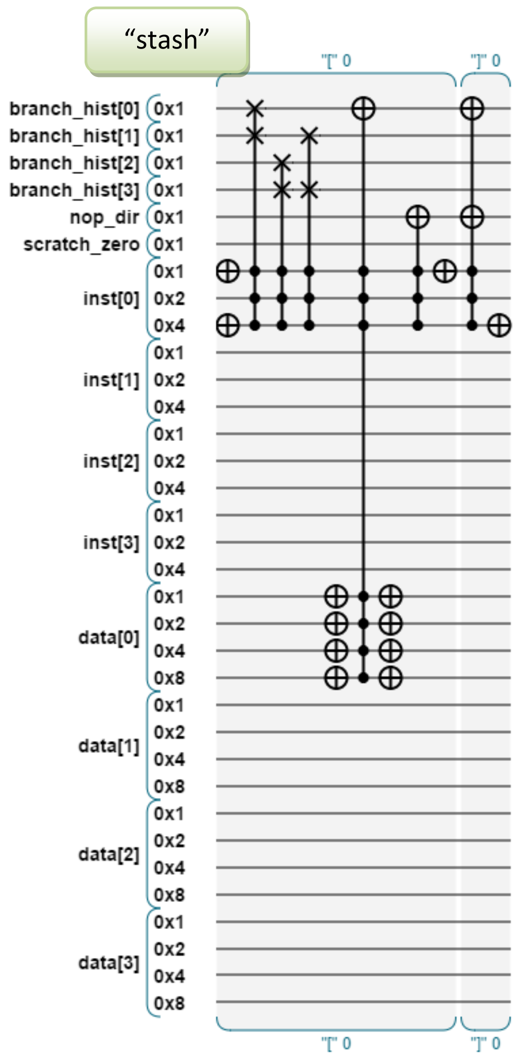
In any case, we can actually implement both of these, and use whichever is best for any given situation. If we're doing a non-quantum bit-sim of this FSM, choosing option 1 allows us to run programs of any length.

Someday I'd like to run this FSM on a quantum electro-optics rig, and it doesn't seem like clearing bits mid-program is even really an option there, so I'll keep option 2 alive, in hopes that QEO systems will have a enough cheap qubits to run an interesting FSM program.

Time to stop putting it off, and just build it. We need a way to store a rotating bunch of bits for the branch history, so adding another QQTAPE sounds perfect.

Here are both implementations, though some explanation will follow.





Here are the new qubits I added to make branching work:

- **branch_hist** – This is the QQTAPE which absorbs the result of each branch decision. The first element (branch_hist[0]) also serves as a “nop” bit. Basically, if we hit a branch instruction and *p == 0, then switch the “nop” on, which will prevent the rest of the instructions from having any effect (so we effectively skip over them).
- **nop_dir** – This indicates which direction the instructions should roll when the nop qubit is set. When nop is <u>not</u> set, we’re always rolling forward.
- **scratch_zero** – I ***really*** didn’t want to cheese out and add a scratch qubit, but I figured I’d get it working first, and then figure out a way to remove this later. Currently the only way I can think of to remove it adds a ton of new gates.

Here’s a walkthrough of the two methods. They only differ in the first step.

1. When a **[** instruction is encountered, zero out the “nop” qubit (which is branch_hist[0]). (The phrase “zero out” should make you say “oh, there’s a reversibility issue”) How this is done depends on which method we’re using.
    a. **clobber**: Write a zero to branch_hist[1] ***always***, even if we’re not on a [instruction. Then if we are on a [instruction, rotate the zero into the nop position. The reason we need two bits is that nop should get a zero bit only at a [instruction, but we can’t know we’re at one without performing a read on the entire inst register. This way, we only have to read one qubit, and we only collapse the part of the quantum

state which tells how many branches were taken. Note that the “clobber” option is not reversible.

   b. **stash** – This is even simpler. We just have a longer QQTAPE for branch_hist, which gets zeroed out at the beginning, and then we rotate fresh ones in at each [instruction. This method is completely reversible and superposition-friendly, but there’s a catch. **If we run out by branching too many times**, then the machine will re-use values which may be non-zero, or even a crazy superposition, and it will malfunction horribly, or at least unpredictably.

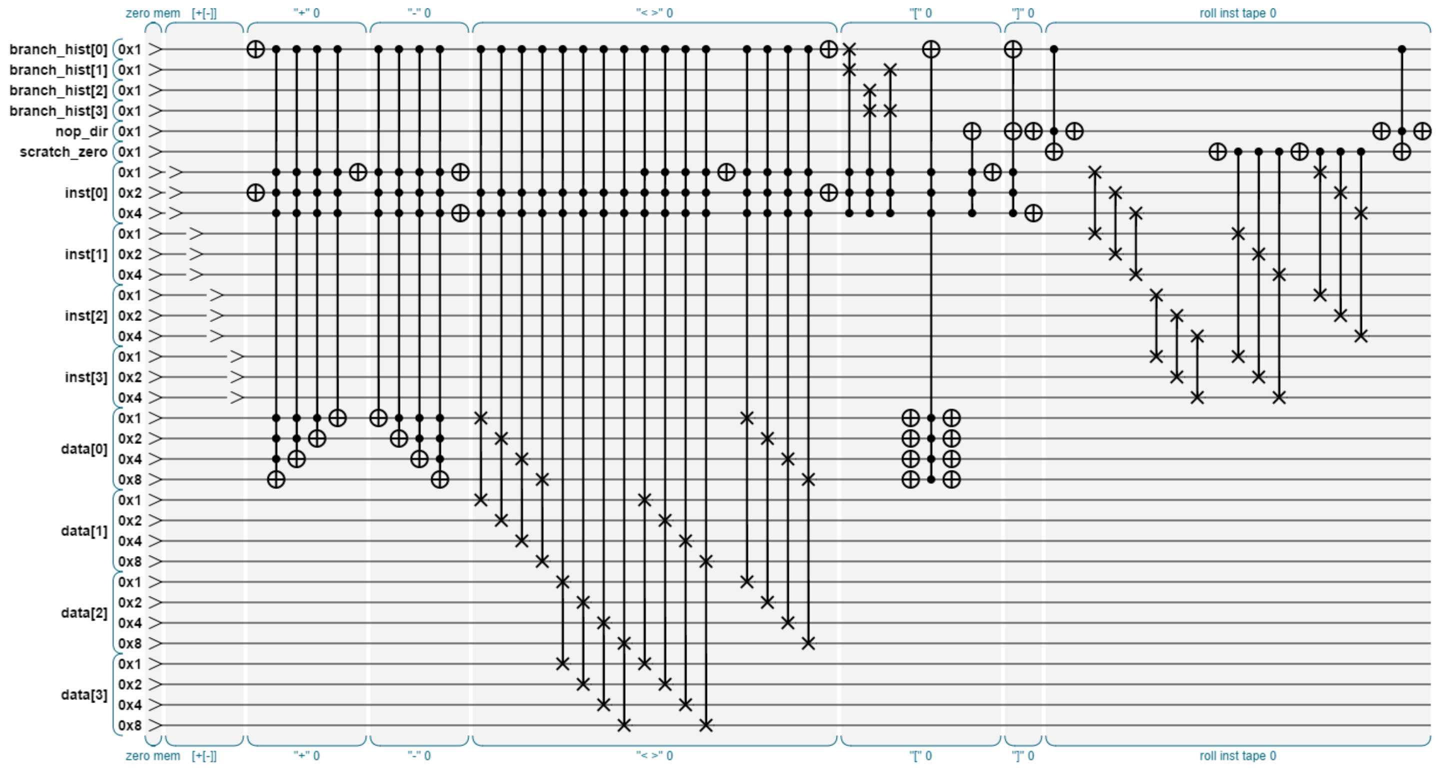
2. Next is the tall gate, which looks like a traffic light. If we’re at a [instruction and all *p bits (that’s data_tape[0]) are zero, flip the nop bit so it’s 1. This indicates that future instructions will be skipped, not executed.

3. Last thing to do for the [instruction is to flip the nop_dir qubit, to indicate that we’ll be skipping forward, if we’re skipping at all.
4. Then, when a] (that’s the end-loop instruction) is encountered, all we need to do is flip both the nop bit and the nop_dir bit, and we’re ready yo go.

Some modifications are needed (of course) to the other instructions:

- For the + - < > . , instructions, add a condition which prevents them from happening if the nop qubit is set.
- For the instruction roll, use the scratch_zero bit (see, I *almost* didn’t need one) to set a flag if nop is set AND nop_dir is zero. Then use the scratch_zero bit to control the roll direction. Note that we set the scratch_zero bit back to zero afterward, so this part is reversible, there’s no waste, and we can even re-use it in other parts of the machine if we need to.

Here’s what the machine looks like so far. It’s almost done, and it’s still not very complicated. Even better, the parallelization looks promising.



Here's the source for this new logic:

```
// [ instruction (loop begin)
if (this.branch_clobber)
  this.branch_hist.storage[1].write(0); // Clobber mode loses superposition, but can run forever.
this.branch_hist.rollfwd(qintMask([inst, this.inst_jmp1]), qintMask([inst, ~this.inst_jmp1]));
nop_flag.cnot(    null, 1, qintMask([inst, this.inst_jmp1]), qintMask([data, -1, inst, ~this.inst_jmp1]));
this.nop_dir.cnot(null, 1, qintMask([inst, this.inst_jmp1]), qintMask([inst, ~this.inst_jmp1]));

// ] instruction (loop end)
nop_flag.cnot(    null, 1, qintMask([inst, this.inst_jmp2]), qintMask([inst, ~this.inst_jmp2]));
this.nop_dir.cnot(null, 1, qintMask([inst, this.inst_jmp2]), qintMask([inst, ~this.inst_jmp2]));
```

So let's run it on the program ++[-]+. This translates to the following C program:

```
++*p;
++*p;
while(*p) {
  --*p;
}
++*p;
```

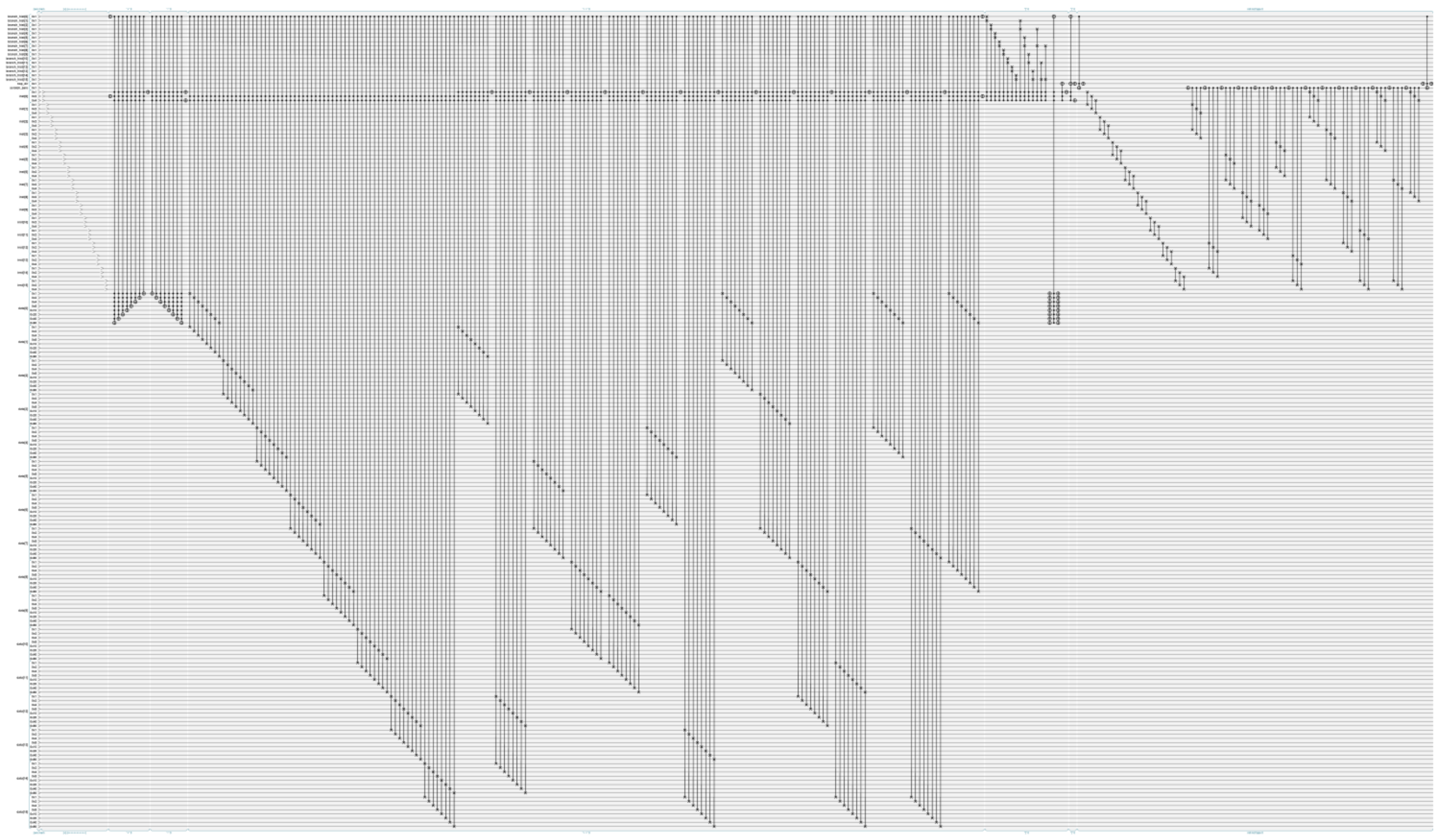
…so if *p starts at zero, it should count up to 2, and then stay in the loop until it's zero again. Here's the run output from loading this program into our current QQFSM, with some commentary:

```
Using 40 qubits.
RAM est: 8388608 MB.
------ FSM tick 0
  inst: + + [ - ] +    // increment *p
  data: 0 0 0 0
  nop: 0 nop_dir: 0
------ FSM tick 1
  inst: + [ - ] + +    // increment *p
  data: 1 0 0 0
  nop: 0 nop_dir: 0
------ FSM tick 2
  inst: [ - ] + + +    // loop, *p!=0, so
  data: 2 0 0 0        // don't set nop
  nop: 0 nop_dir: 0
------ FSM tick 3
  inst: - ] + + + [    // decrement *p
  data: 2 0 0 0
  nop: 0 nop_dir: 1
------ FSM tick 4
  inst: ] + + + [ -    // end loop, flip
  data: 1 0 0 0        // nop and nop_dir
  nop: 0 nop_dir: 1
------ FSM tick 5
  inst: - ] + + + [    // we stepped
  data: 1 0 0 0        // backwards! Yay.
  nop: 1 nop_dir: 0
------ FSM tick 6
  inst: [ - ] + + +    // back at start
  data: 1 0 0 0        // *p!=0, so nop=0
  nop: 1 nop_dir: 0
```

```
------ FSM tick 7
  inst: - ] + + + [    // decrement *p
  data: 1 0 0 0
  nop: 0 nop_dir: 1
------ FSM tick 8
  inst: ] + + + [ -    // end loop, flip
  data: 0 0 0 0        // nop and nop_dir
  nop: 0 nop_dir: 1
------ FSM tick 9
  inst: - ] + + + [    // step back
  data: 0 0 0 0
  nop: 1 nop_dir: 0
------ FSM tick 10
  inst: [ - ] + + +    // *p=0 now, so
  data: 0 0 0 0        // set nop to skip
  nop: 1 nop_dir: 0    // forward
------ FSM tick 11
  inst: - ] + + + [    // don't decrement *p
  data: 0 0 0 0
  nop: 1 nop_dir: 1
------ FSM tick 12
  inst: ] + + + [ -    // loop end, flip nop
  data: 0 0 0 0        // and continue fwd,
  nop: 1 nop_dir: 1    // exiting loop!
------ FSM tick 13
  inst: + + + [ - ]    // all done.
  data: 0 0 0 0
  nop: 0 nop_dir: 0
```

Success! The program looped twice, and then exited the loop when *p reached zero.

Just for fun, here's the machine so far, with 16 data bytes of 8 bits each, room for 16 instructions and a 16-branch stash (a total bargain at 194 qubits):



Step 8: New Instructions! (input/output) , .

(live demo: http://qc.machinelevel.com/qqfsm_examples/qqfsm_06_io.html)

For a non-quantum computer, “output” is clearly defined. When a byte is ready, make a digital copy of it to print it, save it to disk, send it to the internet, or to another program. Oh, and keep the original just as it was, in memory.

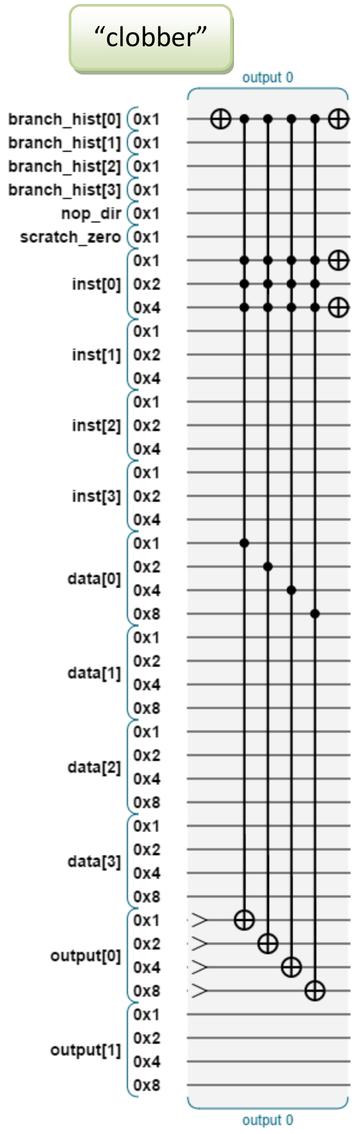
On a QC, that’s trouble. Once superposition and entanglement are involved, printing or digital storage means collapsing the state, which we may not want to do while the program is still running. In fact, making any sort of copy without damaging the original data is going to require fresh qubits.

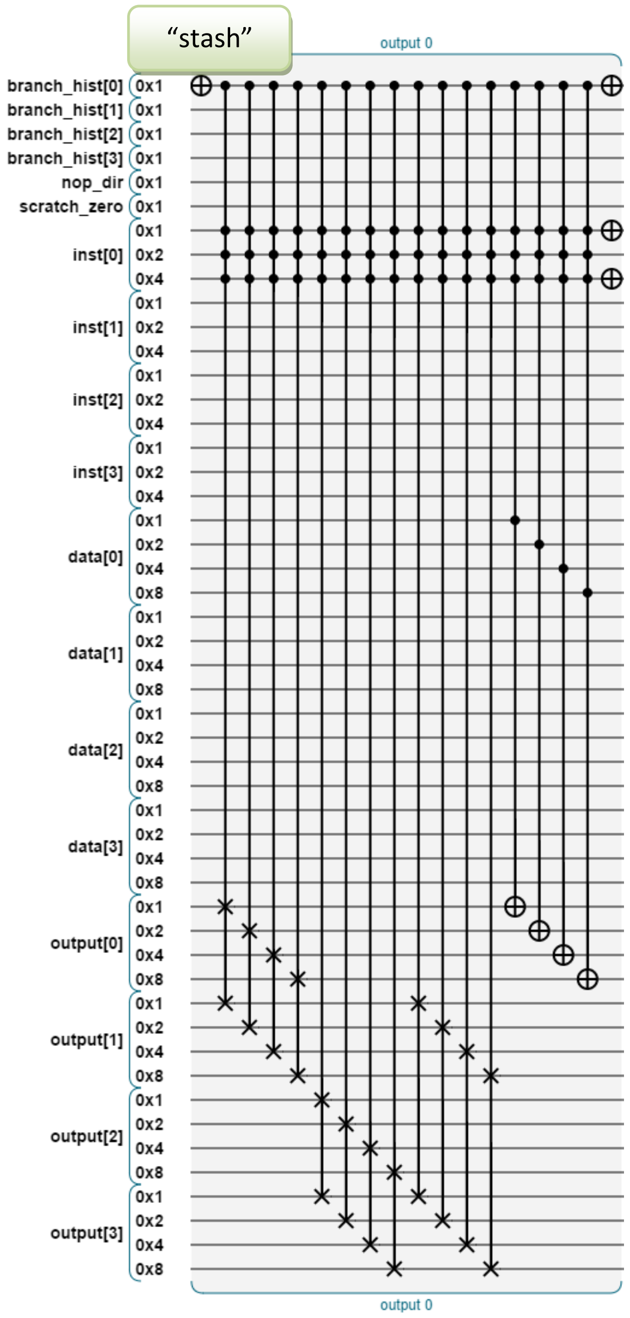
This sounds familiar, so let’s reuse the solutions we came up with last time:

- **clobber** – This time, we can simply zero the output bits every single tick, and then write the new value to them.
- **stash** – Saving up many bytes of output is going to take a big stash, but if we do that, then we’ve got the entire output of our program in spiffy quantum superposition, and we can even hand it off to another program (locally, or via special fiber optics). In fact, we can even teleport these qubits to another QC! Alice and Bob can help.

So clobber will work fine for debugging, but if we clobber everything as we make it, there’s really no point in having a quantum computer.

The two solutions are very different, as you can see here:





Hopefully at this point it’s clear that “clobber” and “stash” would make great names for musicians or bank robbers. Maybe video game characters.

So here’s the source code for this one:

```
    if (this.output_clobber)
      this.output_tape.head().write(0);
    else
      this.output_tape.rollfwd(qintMask([inst, this.inst_out]),
                               qintMask([inst, ~this.inst_out, nop_flag, 1]));
    this.output_tape.head().cnot(data, -1, qintMask([inst, this.inst_out]),
                                      qintMask([inst, ~this.inst_out, nop_flag, 1]));
```

That’s pretty simple, and it handles both cases.

…and here’s the output when we run the program “**++.+..**”:

```
    Using 56 qubits.
    RAM est: 549755813888 MB.
```

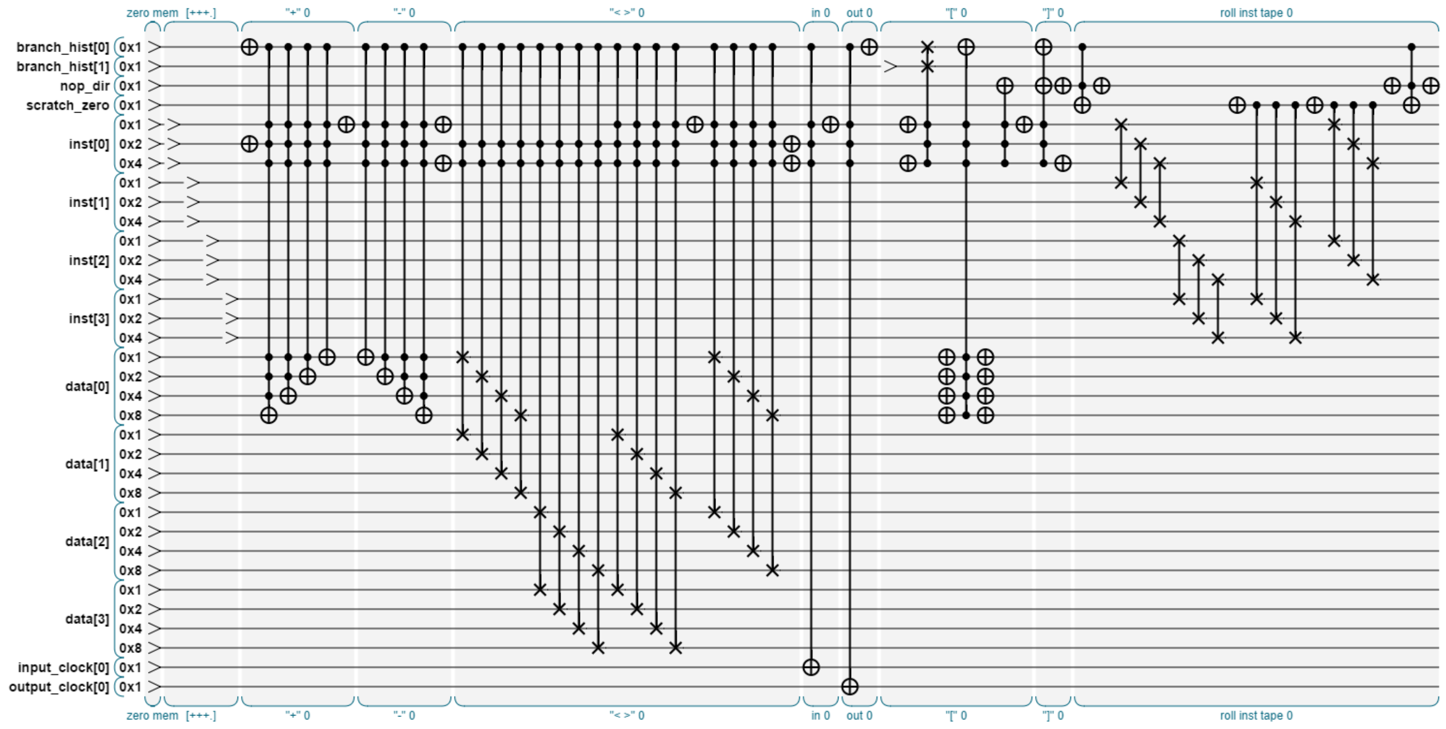
```
------ FSM tick 0
  inst: + + . + . .
  data: 0 0 0 0
  nop: 0 nop_dir: 0
  output: 0 0 0 0
------ FSM tick 1
  inst: + . + . . +
  data: 1 0 0 0
  nop: 0 nop_dir: 0
  output: 0 0 0 0
------ FSM tick 2
  inst: . + . . + +
  data: 2 0 0 0
  nop: 0 nop_dir: 0
  output: 0 0 0 0
------ FSM tick 3
  inst: + . . + + .
  data: 2 0 0 0
  nop: 0 nop_dir: 0
  output: 2 0 0 0
------ FSM tick 4
  inst: . . + + . +
  data: 3 0 0 0
  nop: 0 nop_dir: 0
  output: 2 0 0 0
------ FSM tick 5
  inst: . + + . + .
  data: 3 0 0 0
  nop: 0 nop_dir: 0
  output: 3 0 0 2
------ FSM tick 6
  inst: + + . + . .
  data: 3 0 0 0
  nop: 0 nop_dir: 0
  output: 3 0 2 3
```

Success again! Quantum Computing is fun. I think I've got the output rolling backwards, but it doesn't really matter. Actually, it looks like it may parallelize better this way, just looking at how the gates line up.

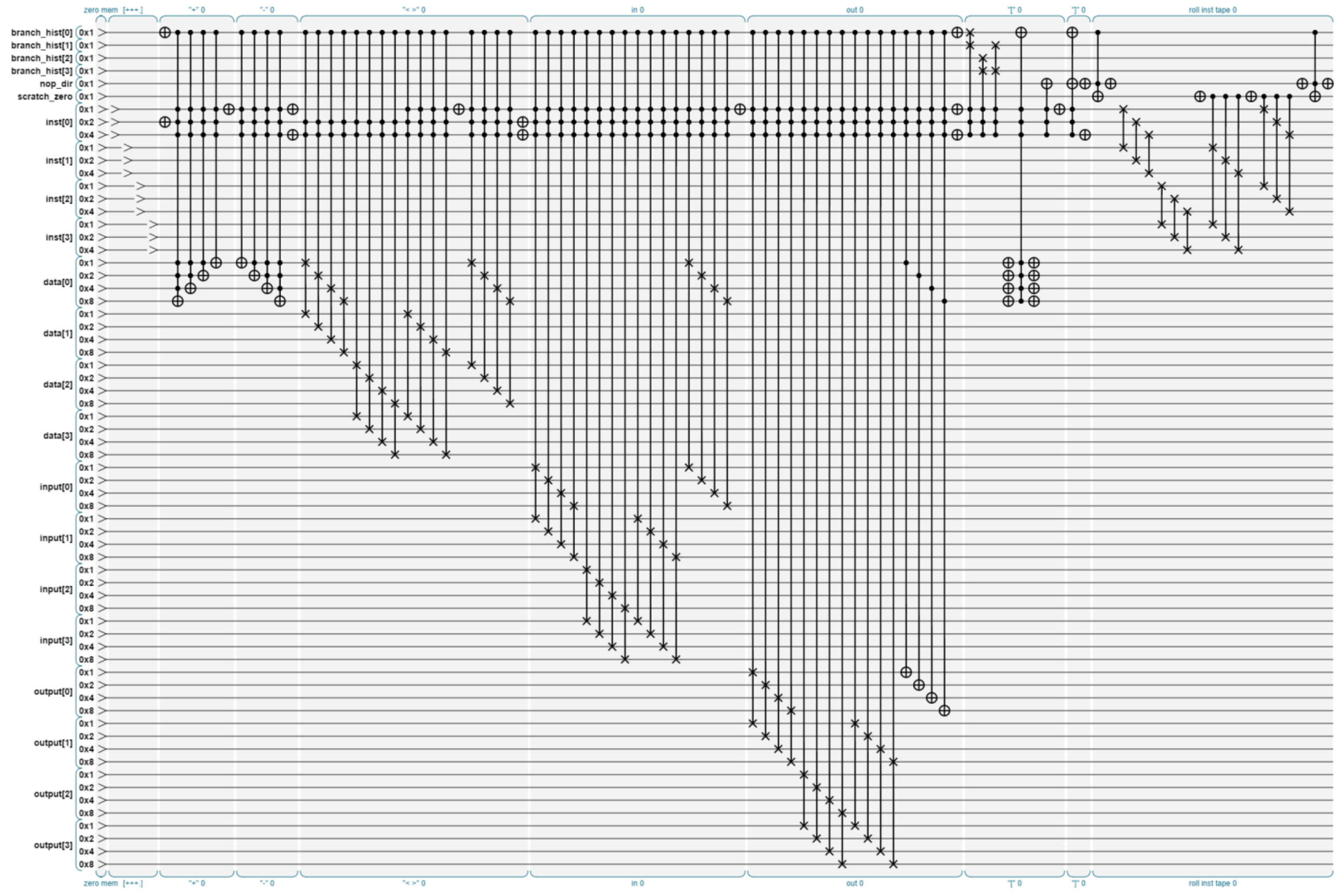
Update: For "clobber" mode, I've decided to change the output so that it simply flips an "output clock" bit, which lets the non-quantum host know that a value is ready to read. Then it reads directly from the register, which eliminates the need for a separate output register.

Then I implemented the "input" instruction using almost exactly the same logic. In "clobber" mode, it simply flips the input_clock bit, signaling the host to write the new value to the data register. In "stash" mode, it simply exchanges the register with whatever's next in the input tape.

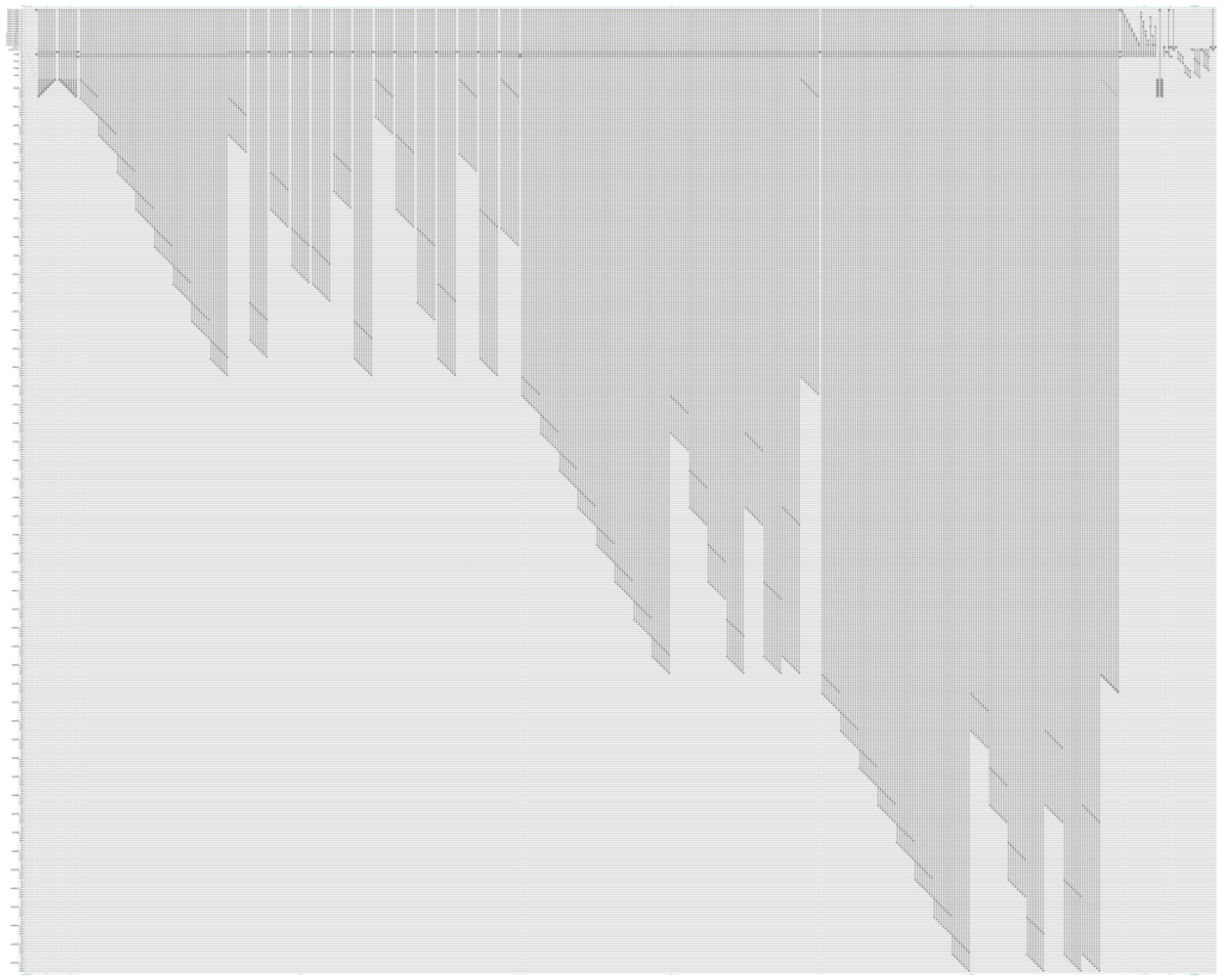
Here's our whole machine so far, in full-clobber mode (34 qubits):



…and here it is in full-stash 4-of-everything mode (66 qubits):



That’s it. That’s what it took us all this time to build, and we should be able to run actual programs on it. Just for fun, here’s the larger-scale one we’ve been playing with (16 of everything, 8-bit bytes, consuming 414 qubits):



Step 9: Surprise! It's Not Another Instruction

There are no more instructions. We've done them all, **and the machine is all finished**. Time to run it (in simulation), and then afterward look into what's needed to actually build a real one.

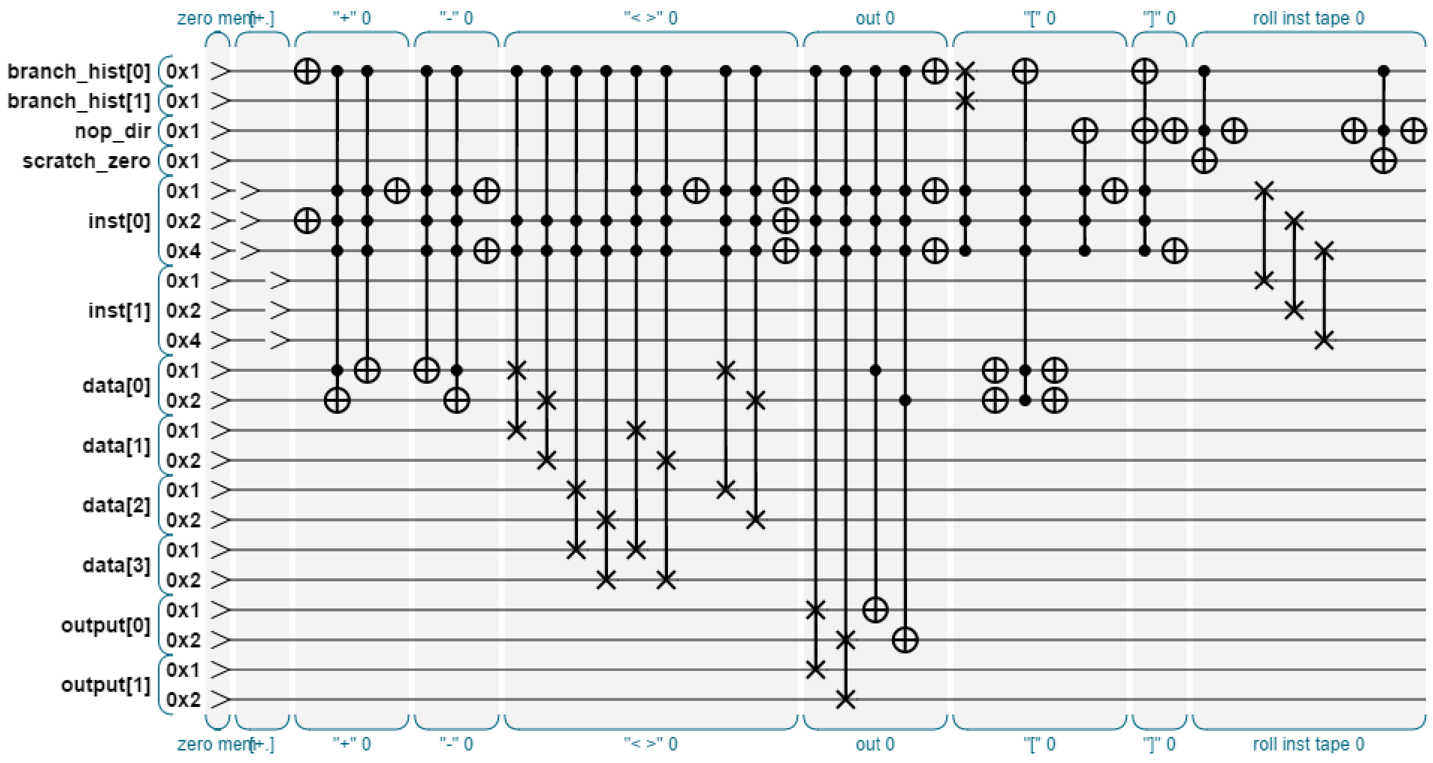
In full-quantum-sim mode, the QCEngine simulator uses 2^n complex numbers to represent the state of *n* qubits. So to simulate the 414-qubit awesomeness in the previous section, we're going to need 2 * 2^{414} floating-point numbers, and we'd better use double-precision for that one, so 8 * 2 * 2^{414} = **2^{418} bytes**. That's about 6.8 x 10^{125} bytes, or 6.2 x 10^{113} terabytes. Considering there are something like 10^{79} atoms in the observable universe, that's going to be expensive.

If we give up the quantum goodness and just simulate it as a digital system (all bits are 1 or 0), then we can simulate the entire state in 414 bits, which is just about 52 bytes. We can do that for sure, but we can't see the quantum effects.

Even the "all-clobber" version in the previous section takes 34 qubits. If we just use single-precision floats, the full-quantum state is still going to take 4 * 2 * 2^{34} bytes = 128 GB. That's much better, but still expensive (and kind of slow) in 2015.

So when it comes time do a full-on quantum sim, we'll simply reduce the number of bytes, bits-per-byte, program length, and maybe strip out features such as input. Each qubit we get rid of cuts the machine size in half, so we can quickly get down to something which can be done on a laptop, or in a browser.

As a quick example, if we strip it *way* down to 4 bytes of 2 bits each, with 2-entry branch and output stashes and no input, then the entire machine fits in 22 qubits. That's 32 megabytes, even as a full-quantum simulation. In January 2015, 32MB is a tiny size, so we probably stripped it down too much. Anyway, here's that version:



...so now what we probably want is two setups: a digital-only simulation of complex programs, and a full-quantum simulation of simple programs.

We have one important limitation: My implementation of the branching does not support *nested* loops. That's okay, for now we'll just run programs which don't require them.

Demo 1a: Non-Quantum "Hello World!"

At the start of this document, I showed an implementation of a "Hello World" program:

```
++++++++++[>+++++++>++++++++++>+++>+<<<<-]>++.>+.+++++++..+++.>++.
<<+++++++++++++++.>.+++.------.--------.>+.>.
```

...so that fits our "no nested loops" restriction well, and it doesn't even require any input at all. I added a simple "host_io" function in the code to harvest the output and print it. Here's the output:

```
Using 383 qubits.
RAM est: 1.5030672529752533e+110 MB.
output (tick 666): H
output (tick 669): He
output (tick 677): Hel
```

```
    output (tick 678): Hell
    output (tick 682): Hello
    output (tick 686): Hello
    output (tick 704): Hello W
    output (tick 706): Hello Wo
    output (tick 710): Hello Wor
    output (tick 717): Hello Worl
    output (tick 726): Hello World
    output (tick 729): Hello World!
    output (tick 731): Hello World!
```

It works! Download a *bf* program off the internet (as long as there aren't nested branches), paste it in, and boom goes the dynamite.

Now for the bad news: TAs you can see in the output, this program (including instructions and RAM) requires 6 bytes of at least 7 bits each, takes 383 qubits and 731 ticks to run. So to try this as a full-quantum sim will take about 1.5x10^{110} bytes, and it will probably take longer than the expected life of the universe to run.

Okay, no problem. Let's pare it down to something smaller. First of all, ASCII is overkill for this demo. We're not using special characters really, so we can offset our alphabet so "a" is 0, "b" is one, etc. Also, we don't need to say hello to the whole *world*, maybe just to everyone looking at the output.

Demo 1b: Non-Quantum "hello"

(live demo: http://qc.machinelevel.com/qqfsm_examples/qqfsm_07_non_quantum_demo.html)

So this program runs with 2 bytes of 4 bits each:

```
    +++++++.---.+++++++>++[<.>-]<+++.[]
```

Note that the first loop (which prints the "L" twice) isn't really needed, it's just there to show off. And there's even a trap at the end so if we run it too long it will stop there, instead of spewing garbage. Here's the output:

```
    Using 121 qubits.
    RAM est: 2.028240960365167e+31 MB.
    output (tick 8): h
    output (tick 12): he
    output (tick 25): hel
    output (tick 35): hell
    output (tick 53): hello
```

That's still pretty neat, and a total bargain at 121 qubits and runs in 53 ticks. We can do a full q-sim in only 2x10^{31} MB. That's still more than my laptop has (don't laugh; remember I'm writing this in 2015.)

Demo 1c: Non-Quantum "hi"

So let's go *crazy* and make the smallest "Hello World" program we can think of. How about this:

```
    .+.-
```

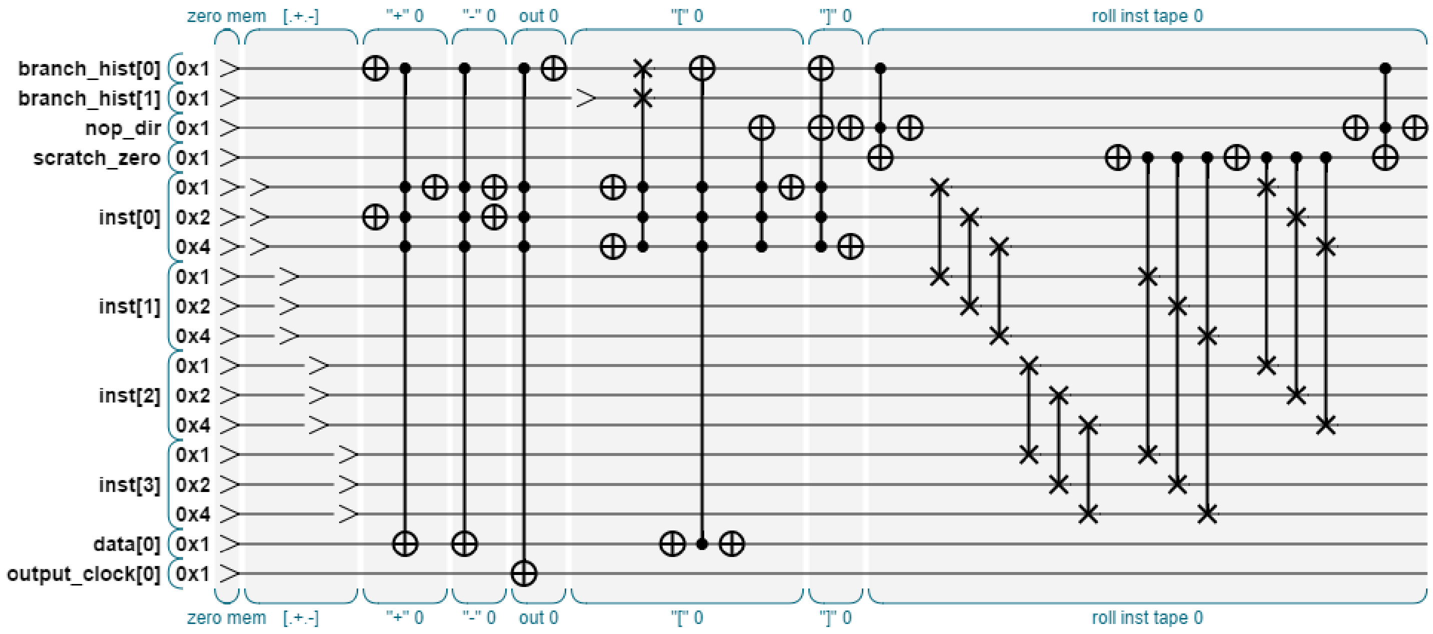
We'll need to use a "h"=0 alphabet, but then it works perfectly, requiring exactly *one bit* of total memory. Okay, maybe we scaled it back too much. Note that the last "-" isn't actually needed, but if we add it then we can run this program extra cycles and it will sound like a happy excited 3-year-old.

Here's the output of a single run:

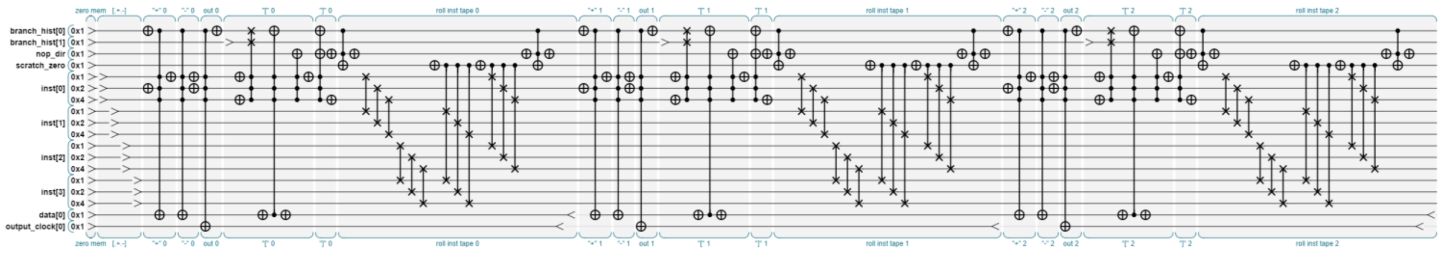
```
Using 18 qubits.
RAM est: 2 MB.
output (tick 1): h
output (tick 3): hi
```

Now we're talking. 18 qubits is easy to simulate, requiring only 2MB. (If you're reading this before 1987, then that's kind of a lot, and I feel for you. Fear not; note that if you remove that last "-" and fix out QQTAPE to handle odd numbers, and only run it for 3 ticks, then we get exactly the same output, but in 15 qubits, which could run in 256k. All good.)

It's not a very interesting machine, but it's a happy one, and that's all we're really ever going for. Here's the circuit for one cycle of this overly-simple machine:



...and here's the full 3-tick run which prints "hi":



It occurs to me that using 2 MB to print "hi" might be setting some kind of a record. Yikes.

Demo 2: Full-Quantum "hi"

(live demo: http://qc.machinelevel.com/qqfsm_examples/qqfsm_08_full_quantum_demo.html)

Now, we finally know we can make program which is not only friendly, but also small enough that we can have some fun with a full simulation. In fact, we're going to go all the way. We'll use "stash" mode instead of "clobber", so the output gets stored up until the end in its full superposition glory. Also, if we add branches, we'll stash those as well.

The actual truth is that when we ran the non-quantum "hi" test above, QCEngine actually ran it in full-simulation mode for us, just because the number of qubits was small-ish. Nothing quantum-weird happened, but only because we only gave it digital instructions.

So now

[I'm still updating this 27 Jan 2015]

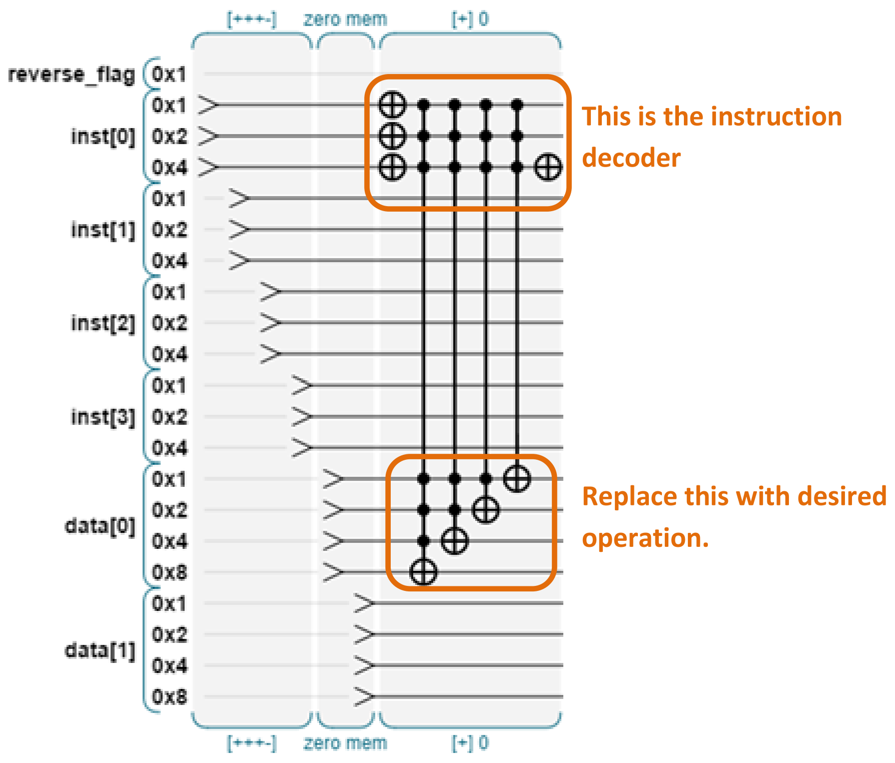
Next Step: Modular Quantum Instructions

What's been demonstrated so far is a full-superposition programmable machine, but one which essentially just performs digital logic on superposed instructions, addresses and values. One benefit is that we can flatten it into a all-digital machine, and it still works.

What would be really great is the ability to add more interesting instructions. While digital instructions (multiply, bit-shift, etc) are straightforward, some native-quantum instructions would be much more fun. We can give this machine a native Hadamard, Rotate, Root-Not, or even something complex but really useful like Invert-About-Mean (used in Grover's Search Algorithm), just as a built-in instruction.

The machine as designed so far only has room for eight instructions, but that's arbitrary. We can add others either by increasing the number of instruction bits, or else by excluding other instructions. Maybe we don't need the input or output instructions, for example.

We already have a generalized instruction format, which we've used for our existing operations. It looks like this:



This expands the horizons of the device substantially. Now we can make instructions to support a computer which is not only Turing-equivalent, but quantum-universal as well. Instead of building custom quantum devices for each function we need to perform, we can do elaborately different programs on the same device, even at the same time.

Game to play: Poison Goblet Penny-Flip

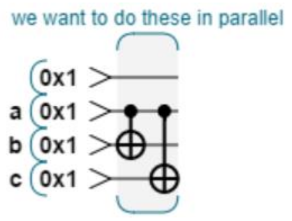
here

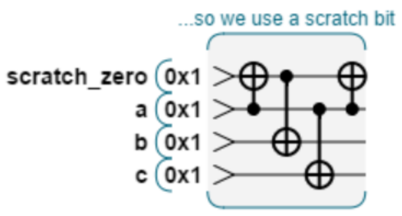
Game to play: Quantum Wumpus

here

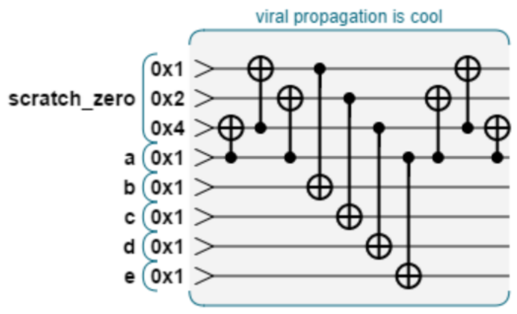
Hardware Limitation 1: Parallelization with Shared Condition Qubits

Most QC hardware is not able to perform two operations simultaneously if *any* of the qubits are shared, either target or condition. We can solve this by adding some spare "zero" bits into the machine, and then temporarily using them as parallel-condition bits as needed. To do this, we c-not the condition bit we want to parallelize, and just be sure to do it again when the operation is done.





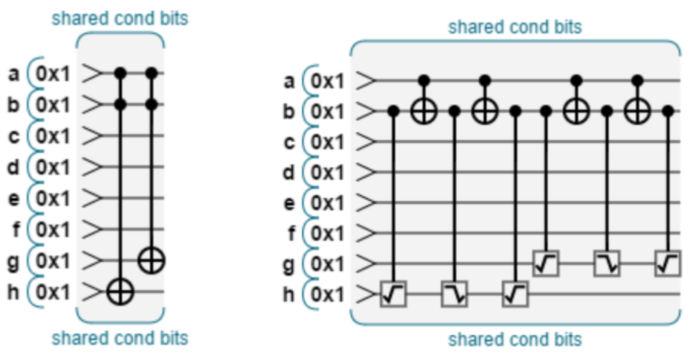
I this example, our operation didn't get any faster, because we ended up with four gates, and only two can be run in parallel. In practice, we can probably do the scratch-prep well in advance (and usually in parallel with some other operation), so we'll often get it for free. In fact, when we need many of these scratch bits at once, we can use them to create more, like zombies and vampires.



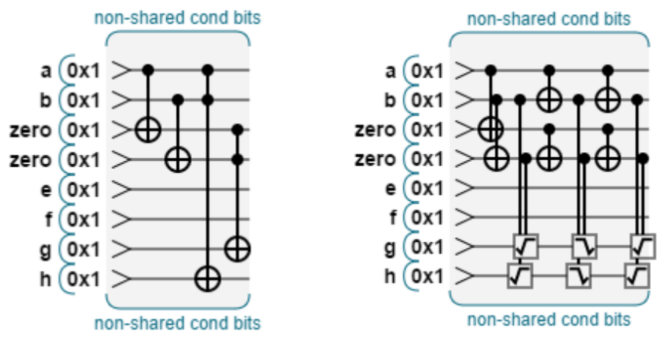
Hardware Limitation 2: 2-Qubit Operations Only

Here's a significant limitation: most current physical QC's can only support a maximum of two qubits interacting in a single gate. That means operations involving more than two qubits need to be broken down into (usually many) smaller operations.

Consider the following gates. Each of them uses three qubits, but we can break them down into 2-qubit operations, like this (see this paper on Elementary gates for QC, section 7.2):



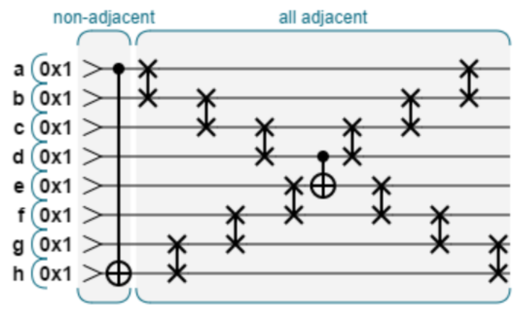
The square-root gates are “root not” and “reverse root not”. This works, and doesn’t add any qubits, but unfortunately it takes the running time from 2 cycles to 10 cycles. There is something we can do to improve this. As mentioned in the previous section, if we add some “waste” qubits, and use them to spread out the condition bits, theings parallelize buch better:



Because we’re able to do those operations in parallel, we finish in 6 cycles instead of 10. We needed two extra qubits, but got almost a 2x speedup.

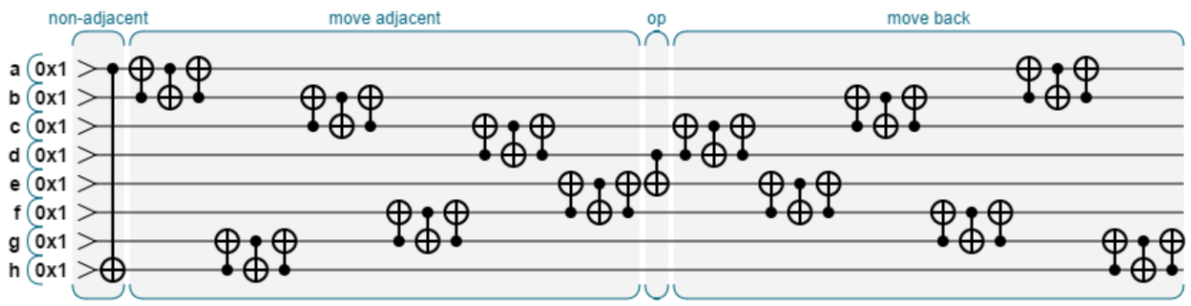
Hardware Limitation 3: Adjacent Qubits Only

This one is my personal favorite. As far as I’m currently aware, ***all*** physical QC devices currently require qubits to be physically adjacent to one another for any interaction to occur. So assuming our qubits are arranged in a line (as we’ve been doing in all of our gate diagrams), we actually need to fix any non-adjacent by using exchanges to get the gates next to each other, like this:



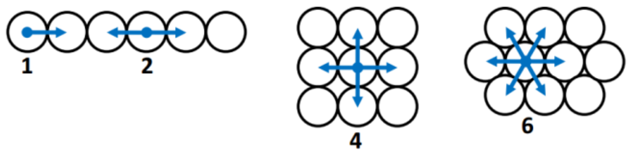
Note that after the operation we *could* simply leave everything where it is, that won’t change the result as long as we’re okay with the bits being in new places. In fact, it’s likely the next operation is going to need them in new positions.

Photonic systems usually have a native “exchange” operator. If the hardware doesn’t have that, then the exchange is implemented with three CNOT gates, so the above example actually looks much worse:



Fortunately, some of these can be done in parallel with each other. Still, we've replaced one "theoretically ideal" instruction with about 37 "physically real" instructions.

Suppose we *don't* actually need them to be in a row, but could store them in a quad or hex grid, like this:



By increasing the number of adjacent neighbors, we should be able to reduce the total amount of travel, and the total complexity of the actual implementation. I explore that a bit in this whitepaper.

Conclusion

[to do].

Useful References

- Minds, Machines, and the Multiverse, by Julian Brown
- Elementary Gates for Quantum Computation (1995) arXiv:quant-ph/9503016v1
- Quantum Computing for Computer Scientists (Yanofsky and Mannucci), 2008
- IBM QC advances 2/28/2012:
    - http://ibmquantumcomputing.tumblr.com/
    - http://www.nytimes.com/2012/02/28/technology/ibm-inch-closer-on-quantum-computer.html
- [TODO: add other papers and books I've found useful]

About the Author

EJ and his muse live in a secret laboratory in San Francisco. Everything else you really need to know is either posted here, or can be obtained by sending an email to ej@machinelevel.com.