1 Background: What is an FPGA, and how is it useful for emulating a quantum computer?

An FPGA is a Field-Programmable Gate Array—in other words, it is a small device similar to an Arduino or other type of microcontroller, but instead of performing sequential code, it builds a circuit of logic gates that can perform combinational logic. To do this, FPGA code is written in one of two somewhat similar languages: VHDL or Verilog (or SystemVerilog, more on that later). These languages have some higher-level capabilities such as different types of loops and conditional logic, but are mostly written using lower-level structures such as wires and logic gates.

To understand why this is useful for emulating a quantum computer, we must first understand how a quantum computer works. In a classical computer, data is made up of strings of bits, which can either take the value 0 or 1. By putting many bits together, we can have many different possibilities for the overall state of the string—2^N different states, to be exact. A quantum computer, on the other hand, is composed of qubits, which are like classical bits except for two important properties: they can be in a superposition of both 0 and 1, and they can be entangled with other qubits. From these properties, we find that a system of N qubits still has 2^N possible states, but rather than being in just one of them at a time, the system can be in all of these states with some complex probability amplitude for each.

So how does this all relate to an FPGA? We can represent a quantum state of N qubits with a vector containing 2^N complex numbers each representing the probability amplitude for that state. To operate on these states, we need quantum gates, which we can represent with a matrix of size $2^N * 2^N$. We can multiply the gate matrix by the state vector to get the output state of the operation. The problem this poses for classical computers is that as N increases, the number of complex numbers in the state and gate get vastly larger, and when it comes time to multiply the matrix by the vector, the computer has to step through each element in the matrix sequentially. For a system of 20

qubits, if a computer could perform the necessary operations on 1 billion gate elements per second, it would still take around 20 minutes to get through the whole gate—and that's just one gate. Most quantum algorithms require circuits with at least a few gates, and that either requires multiplying the matrices together or operating with them one after the other, which both drastically increase the runtime. However, due to an FPGA's circuit-like nature, we can build circuits that will perform this operation instantly, provided we have space on the FPGA to build them.

2 How do we represent numbers on the FPGA?

It is easy to convert strings of bits into integers, simply by converting from binary to decimal. However, the numbers necessary for this are generally not going to be integers, and this requires a bit more consideration of size and precision. I ultimately settled on using 16-bit fixed point numbers, to provide a balance between these two factors. In a fixed-point number, unlike a floating-point, the decimal point is in a fixed place; everything before it is treated as an integer (except sometimes the first bit, which can be used to specify the sign) and everything after it is treated as a negative power of 2, i.e. a fraction. The first bit of the fractional part is 0.5, the second is 0.25, and so on. It just so happens that, in a quantum computer, due to states being normalized and gates being unitary, we never actually have to use a number greater than 1. Thus I decided to use the first bit as the sign, the second as the integer, and the remaining 14 bits for the fractional part. This provides a sufficient degree of precision (for $1/\sqrt{2}$, an irrational number, this representation is only 0.002% off) while being only a fourth of the size of a typical floating-point number.

Since we are using bits to represent fixed-point numbers, we have to have some way to perform arithmetic on them. I used a fixed-point arithmetic library for Verilog found here: Overview :: Fixed Point Math Library for Verilog. I only needed the qadd and qmult modules for my purposes, and it is worth noting that in both, the output number is the same format as the input numbers. This

makes it even more important that we know none of the numbers will be greater than 1, so we can avoid overflow.

To represent complex numbers, I decided to use the typedef struct keywords in SystemVerilog (not exactly the same as regular Verilog, but they work when used together) to define a data type containing two 16-bit numbers: one for the real part and one for the imaginary. With this, I was able to define states and gates to be arrays/matrices of complexNums to make things simpler, and access the real or imaginary parts of each complexNum accordingly.

3 Gate * state multiplication

This is the most significant reason for the long runtimes on a classical computer trying to emulate a quantum computer, so it is the most important part to move onto the FPGA. Since arithmetic is done through modules, not simple operators, we need a way to build a circuit of as many modules as is needed for a given number of qubits. Fortunately, we can do this in a SystemVerilog generate block. This type of block is a loop that runs during synthesis (i.e. when the compiler is trying to figure out how to build a circuit on the FPGA that implements your design) rather than runtime, so it creates the appropriate structure of multipliers and adders. The structure is as follows:

Loop through each row of the gate, and within each row, loop through each element in the row. For each element, multiply it by the corresponding element in the state and store the result in a temporary array. Since both numbers are complex, we will need to use 4 multipliers to FOIL the numbers. Also, after multiplying the imaginary components, we must negate the result due to the implied i^2—this is done by simply flipping the most significant bit and keeping the rest the same, storing this new result in a new part of the array. Finally, we must add the real and imaginary parts separately, storing their sums in another temporary array of complexNums. The final result for each element is $(a_1 + b_1 i)(a_2 + b_2 i) = (a_1 a_2 - b_1 b_2) + (a_1 b_2 + b_1 a_2)i$.

After looping through each column in a row, we must add up all the real and imaginary components in the row to get the final value for that element of the output state. This is a bit tricky since we have 2^N things to add, and can only add them two at a time. But since this is a power of two, we know that if we keep adding two at a time, then adding their sums, etc., eventually we will have just one final sum. For example, with 3 qubits we have 8 elements to add, so we can start with 4 adders: 1+2, 3+4, 5+6, 7+8, then 2 adders for their results: (1+2)+(3+4), (5+6)+(7+8), then one final adder for the total sum: (1+2+3+4)+(5+6+7+8). By doing this for both real and imaginary components, for each row in the gate, we have our output state calculated. Again, since this is a circuit built in hardware, it is "evaluated" immediately as soon as the correct inputs are loaded in, rather than performing all these calculations sequentially as a classical computer would.

4 Gate * state multiplication - real numbers only

All these multipliers and adders take up quite a bit of space, and on my FPGA board I was only able to fit a 2-qubit full multiplication module in my emulator. However, there are a number of possible gates and states that don't have any imaginary component, so at the cost of some generality, we can greatly decrease the size of the multiplication module. This adapted version of the above module performs most of the same steps, except for each element we now only have 1 multiplier rather than 4 multipliers, 2 adders, and a NOT gate. With this assumption, I was able to fit a 3-qubit emulator on my FPGA with quite a bit of room left over. The top SystemVerilog module in the emulator has two possible multiplication module instantiation blocks, allowing the user to comment out the one that is not needed before synthesis.

5 UART communication

We now have a way to operate with a gate on a state on the FPGA, but how do we define what those gates and states are? This is something that has to be done on the PC (with QuTiP—more on that later), so we have to have some way of communicating between the PC and the FPGA. This is where UART communication comes in. Unfortunately, this communication can be pretty difficult and complicated. Vivado comes with some IP cores that assist with this, and I decided to use a MicroBlaze MCS since it had all the functionality I needed while being simple enough to not take up much space. The code for the MicroBlaze has to be written in C, and can be associated with the Verilog code before Vivado generates the bitstream (thus changes to the MicroBlaze code take some time, but not as long as full synthesis). The MicroBlaze acts as a sort of "middleman" between the PC, which is creating/sending the state and gates and receiving the output, and the FPGA, which is computing the output. Here's how the whole chain of communication works:

First, the Python program opens the serial port where the FPGA is plugged into the PC at 115200 baud. After creating the initial state and gates, it steps through each element and separately converts each of the real and imaginary parts into 2 bytes representing 16-bit fixed-point numbers. It sends the two bytes for each number through the UART. While it is doing this, the MicroBlaze is waiting to receive each single byte, and when it does, it writes it to a channel that the FPGA can access. After receiving both bytes of a real number, it sets a flag on a different channel high so that the FPGA knows it can now access the number. It then repeats this to receive and write two bytes of an imaginary number, setting the flag low and moving back to the real number. The only other aspect of sending the state/gates is being able to send an unspecified number of gates through. This is done by sending a byte that is all 0s before sending the last gate in the sequence, and a byte with a 1 at the end for all the previous gates. From this byte, the MicroBlaze and FPGA know whether or not they should repeat the gate-receiving process.

When all of the values for the state and gate have been sent and loaded to the FPGA, the output is ready to be sent back to the PC. This is done in essentially the same way as before, but in reverse. For each element, the MicroBlaze sets a flag high or low before reading the two bytes from the FPGA. After reading each one, it prints it to the serial console. Python, now waiting to receive a certain number of bytes back from the MicroBlaze, reads each line, strips off newline and return chars, and converts two bytes into their floating-point representation. It loads each number into the appropriate place in the output state, and when this is done, it has the full output state ready for measurement.

6 FPGA finite state machine

While the PC and MicroBlaze are communicating, the FPGA is going through its own finite state machine in order to know when (and what) to read from the MicroBlaze. A finite state machine is essentially a program that moves through a series of distinct steps, or states, in a certain way based on conditional logic checks, sort of like a flowchart. This is useful for implementing a sort of sequential code on an FPGA, which typically lacks sequential capability, and breaking it into chunks that can be repeated or skipped as needed.

To do this, the FPGA top module contains a 3-bit state register that specifies which of the 8 states the FPGA is in. It contains an always @(posedge clk)block, meaning that the FPGA will repeat the code contained within each time the onboard clock goes high (for the Basys3, every 10 ns). Within this is a case block which checks the value of the state register and performs the specified actions for that state. Generally, each state must do two things: update the value of all variables used within the always block (even if they don't change) and decide which state to move to next. The states are as follows:

Starts in RESET state - sets the row and column counters to 0 and moves to LOAD_STATE_REAL.

LOAD_STATE_REAL - waits for the load_ready flag to go high, then loads the number into the real component of the current index of the input state. Then moves to LOAD_STATE_IMAG.

LOAD_STATE_IMAG - waits for the load_ready flag to go low, then loads the number into the imaginary component of the current index of the input state. If the index has reached its max value $(2^N - 1)$, the state is finished loading, so it moves to LOAD_GATE_REAL and resets counters. If the index is not yet at max value, the state is not finished, so it increases the index by one and goes back to LOAD_STATE_REAL.

LOAD_GATE_REAL - similar to LOAD_STATE_REAL, waits for load_ready to go high, then loads the number into the real part of the appropriate position in the gate (row, col) and moves to LOAD_GATE_IMAG.

LOAD_GATE_IMAG - waits for load_ready to go low, then loads the number into the imaginary part of the appropriate position in the gate. If it has reached the end of the gate (both row and col are at $2^N - 1$), it resets the counters and moves to CHECK_REPEAT. If it has reached the end of a row but not the last row, it resets col to 0 and adds 1 to row, then goes back to LOAD_GATE_REAL. If it has not yet reached the end of a row, it adds 1 to col and keeps row the same, then goes back to LOAD_GATE_REAL

CHECK_REPEAT - once a gate is finished loading, reads the 0th bit of the new_gate flag. If it is high, there is a new gate coming in that is now operating on the output state of the previous operation, thus it loads the current outState into state and move back to LOAD_GATE_REAL with indices reset. If it is low, there is no new gate, so it moves to SEND_STATE_REAL.

SEND_STATE_REAL - works in the same way as LOAD_STATE_REAL, except now it sends the real part of the current number to the MicroBlaze so it can send it to the PC and moves to SEND_STATE_IMAG.

SEND_STATE_IMAG - works in the same way as LOAD_STATE_IMAG, except now sends the imaginary part of the current number to the MicroBlaze. When it is finished, it goes back to RESET, which then goes to LOAD_GATE_REAL to await the next state/gates sent by the PC.

In addition to the infrastructure for this finite state machine, the FPGA has to create the multiplication module and MicroBlaze, and these 3 components make up the top module on the FPGA.

7 State measurement

The final component in emulating a quantum computer is emulating the process of measuring the output state. This is a sort of operation on a state, but there is no simple gate that represents it, so we must design some other way to perform measurement. This is the most significant divergence in my emulator design from the paper that inspired me (An FPGA-based real quantum computer emulator), as I elected to perform measurement in Python rather than on the FPGA. My reasons for this were twofold: firstly, it involved actions possible in VHDL that I'm not sure are possible (or as simple) in Verilog; secondly, and more importantly, I wanted to make a hybrid PC/FPGA emulator where everything having to do with gates was done on the FPGA, and everything else was done on the PC. Since gates increase in size much faster than states, I hoped that this would free up enough space on the FPGA to fit a full 3-qubit emulator, while not incurring a significant increase in runtime. Ultimately, I did not make quite enough space to fit a 3-qubit emulator on my board, but I did clear up some space. The method for measurement is as follows:

Choose a qubit to measure (the ith qubit) in a state representing N qubits. For each element (probability amplitude) in the state, convert the index of the element into an N-bit binary number representing the state associated with that probability amplitude. For example, the element at index 3 in a 3-qubit system is the probability amplitude for the state $|011\rangle$. We want to find the

total probability to measure a certain qubit to be 0, so if the ith bit in the index is 0, add the probability amplitude mod squared to a total value. In the example above, if we were measuring the first qubit, we would count that probability amplitude because the first qubit in that state is in the $|0\rangle$ state. When we have found the total probability to measure the ith qubit to be 0, generate a random number and compare it to this total. If the random number is higher than the probability, this means we have measured the ith qubit to be 1, so we loop through the state and erase all probability amplitudes where the qubit is 0. If the random number is lower than the probability, we have measured the ith qubit to be 0, so we erase all the amplitudes where it is 1. After this, we must normalize the state again. Once this is done, we have the final measured state.

8 Python front-end

To use the emulator, we simply must create the input state and gates in Python so they can be sent to the FPGA. I chose to do this using QuTiP (Quantum Toolbox in Python), since it has the capability to quickly create many different kinds of quantum gates in all sorts of configurations, take tensor products, and other useful operations for creating whatever gates are needed. In the main body of my Python script, I have a block to declare the input state (and normalize it, if it is not already) and a block that creates an array of gates and appends the necessary gates to it, according to the order in which they will operate on the state. The actual process of sending the gates to the FPGA and receiving the output is contained within the emulate() method. This allows the emulator to run multiple times with the same gates and input state without having to create those gates from scratch, typically saving about 0.1 s.

9 Instructions for using Vivado with Basys3 FPGA board

1. Install free version of Vivado from Downloads (use version 2019.1, as 2019.2 gets rid of the SDK mentioned below). Download and installation both will take a long time and take up a lot of disk space. During installation, select option that includes SDK.

2. Request license and copy into any folder, then click 'Copy License' in license manager and activate the license.

3. Follow instructions at Vivado Version 2015.1 and Later Board File Installation (Legacy) [Reference.Digilentinc] to install board support files for Basys3 board.

4. Restart Vivado and when it opens, click create new project. Select RTL project and make sure 'do not specify sources at this time' is selected. On the next screen, click on boards at the top and search for the Basys3 board, then double click on it (not on the text, this will open some other link). Click Finish to create project.

5. In Project Manager on the left, click Add Sources, then add or create design sources. On the next page, click Add Files and add appropriate .v or .sv file, then click Finish.

6. Download Basys3_Master.xdc from Basys3/Resources/XDC at master · Digilent/Basys3 · GitHub. Then go back to Add Sources, this time selecting add or create constraints, and add the .xdc file.

7. If using a project with a microblaze_mcs processor module (you can open the .v or .sv file in Vivado and look through it to see if there is one), you will need to generate the core for that as well. Click on IP Catalog under the project manager, then search for MicroBlaze MCS and double click on it. You will need to generate the core with the specifications in the module definition. On the MCS tab, change the memory size to 32 KB. On the UART tab, enable the receiver if there is an RsRx connection in the module definiton and the transmitter if there is

an RsTx connection. For the GPO and GPI tabs, in the module definition, GPI is represented by the .GPIOX_tri_i and GPO by the .GPIOX_tri_o lines, where X is the channel for each. Add the required number of GPOs and GPIs with the number of bits set to the size of whatever variable it is assigned to (e.g. if GPO1 is assigned to led and led is 8 bits wide, then GPO1 must be 8 bits wide). If there is an interrupt in the module definiton, enable a rising edge interrupt on the appropriate GPI (this will usually be one that is only 1 bit wide). Click Finish and let it run to generate the IP.

8. For code containing a MicroBlaze, there will also be a C file for the processor. To use this, first click Run Synthesis on the left, then once that finishes, go to File→Export→Export Hardware and click ok. Then go to File→Launch SDK and create a new application project. In the hardware platform section, click New, then Browse, then find the .hdf file you just created. It should be in the .sdk folder in the Vivado project folder. Back on the create project screen, make sure the language is set to C and click Finish.

9. Open the default helloworld.c file (in [project name] → src on the left). Then open the appropriate helloworld.c from my GitHub and copy its contents into the default helloworld.c. Save and it should generate a .elf file automatically. Back in Vivado, go to Tools → Associate ELF Files and click on the 3 dots next to the top associated ELF file (the one under Design Sources, not Simulation Sources). Find the .elf file for the SDK project you just made (it should be in the .sdk folder → [project name] → Debug) and click ok to associate the ELF file.

10. Under Program and Debug at the bottom of the left panel on Vivado, click Generate Bitstream and let it run. When it finishes, select Open Hardware Manager and hit ok (or just cancel if the hardware manager is already open).

11. In the Hardware Manager, below Program and Debug, click Open Target with the FPGA plugged in. It should connect automatically to the FPGA. Finally, under Open Target, click

Program Device, then click on xc7a35t_0.

12. To monitor what the FPGA is sending/receiving from the serial console, I use PuTTY, but the SDK also should have a built in serial console. At the bottom, click on SDK Terminal, then the green plus button. Add the appropriate port and baud rate and it should allow you to view the serial communications.

10 Instructions for running emulator

1. Follow steps in Vivado instructions to set up quantumCompFSM.sv as the top module in a project, import other .sv and .v source files and .xdc contstraint file, and set up MicroBlaze with helloworld.c code. If using full complex number generality, make sure the gateStateMult module block is uncommented and the gsmRealOnly block is commented, and N = 2. If using only real numbers, comment out the gateStateMult block and uncomment the gsmRealOnly, and set N = 3. When bitstream with associated ELF file is written, open the hardware manager and program the device.

2. Once bitstream is uploaded, the FPGA should be ready to receive state and gates from running the emulate() method after running gateSender.py with the appropriate gates. It will print 4 things: first, the raw data received back from the FPGA converted to floating point numbers; second, that data converted to a normalized state; third, the output of the measurement of that state; and fourth, the runtime of the method.

3. To change/add gates: use QuTiP commands to append each new gate to gates list in gateSender.py.

4. To change number of qubits: change N in quantumCompFSM.sv, numQ in helloworld.c and gateSender.py. SV code will need to be re-synthesized/implemented and a new bitstream will need to be written each time the number of qubits is changed. As of now, there is not enough

space on the FPGA for more than 2 qubits (or 3 with only real numbers), and any gate that is designed to operate on fewer than N qubits can be expanded to N qubits by simply taking the tensor product with the identity for the qubits that are unaffected. Because of this, I would just leave it always synthesized for the max number of qubits.