Beamforming

Directing a Signal

Peter Isho
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
pdisho@oakland.edu

***Abstract*—Beamforming is a way of directing a signal to the users will. This is achieved by matrix manipulation which won't be discussed here due to its complexity. In this case, only one portion of the beamformer is successfully working. The entire beamformer has been developed, but isn't fully functional.**

I. Introduction

A beamformer will take an input signal from any direction and output in a desired direction. This project has many implications in industries where signals are everywhere. One good example is with a basic Wi-Fi router which outputs Wi-Fi in all directions. With this beamformer, Wi-Fi can be sent directly to a device that needed it. This would allow for a faster, stronger, and longer-range signal since it can be focused in one direction instead of everywhere. Wi-Fi is one example, but this idea can be applied in many industries that deal with signals.

This report will mostly contain information about one part of the beamformer because that is the part that is fully finished. A beamformer is consisted of three different "cells" which include an internal cell, a boundary cell, and a final processing cell. They are connected in a cascading fashion where the outputs of one cell are the inputs of another cell. The boundary cell is the most complex one and is the one that will be discussed in detail here. Everything that is discussed is on the FPGA with no external peripherals. This is how an ideal beamformer would be with some I/O's as needed for data acquisition. Discussion about the internals of the design include pipelining, timing, cordic, division, multiplication, and feedback loops. A two antenna beamformer has also been developed, but it's not working correctly so it won't be discussed in detail.

II. Methodology

A. Boundary Cell

This cell consists of three inputs, four outputs, and four internal registers that are updated every iteration. The signals that will be discussed can be seen in Figure 1. The related equations are below figure 1 and will be the main topics of this report. They seem to be basic equations, but implementing everything on an FPGA isn't as trivial as it seems. Timing was necessary to perfect so that every signal arrives to components when they are supposed to.

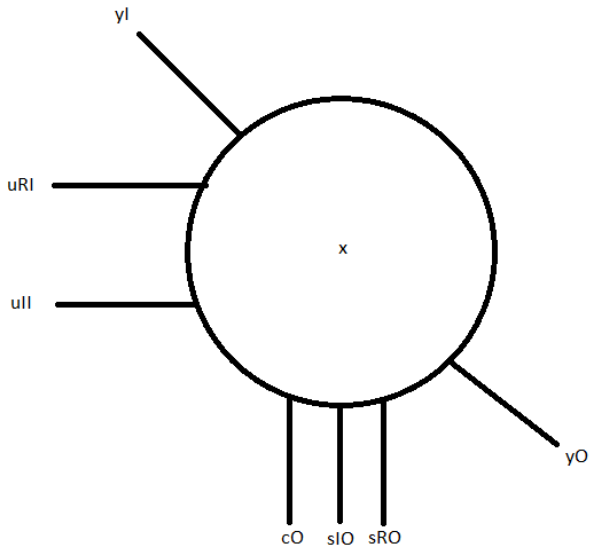

Figure 1. Boundary Cell IO

$$x' = \sqrt{x^2 + |uI|^2} \quad [1]$$

$$cO = \frac{x}{x'} \quad [2]$$

$$sO = \frac{uI}{x'} \quad [3]$$

$$x = x' \quad [4]$$

$$yO = cO * yI \quad [5]$$

One thing to be noted is that uI and sO are complex numbers, that is the reason for two separate signals in Figure 1. The best way to deal with complex numbers in hardware is to have two signals for the real and imaginary parts. A second way would be to have the most significant bits relate to the real portion while the least significant bits relate to the imaginary portion. This method seems much more complicated so method one was used in this project.

To start with the first equation regarding x', at first this equation seems incredibly difficult to implement in hardware. After some time, it was noticed that this is the exact output of Xn in a cordic component. This relation made this project manageable because it is known that square root is complicated to implement in hardware. However, that equation can't be completed first because the absolute value of a complex number is the square root of the

squared roots. This means that a signal, absU, needs to be calculated using the same Cordic component.

The first thing that is done when start goes high is to calculate absU with the inputs uRI and uII. Those two signals go into a cordic and after about 20 cycles, the Xout of cordic is the value of absU * An. An is a constant which, depending on whether using the normal or expanded version of cordic, is 1.647 or 3.294, respectively. Since the output of Xout is multiplied by that number, it makes sense to divide by it to achieve the true value of absU. Division, however, is one of the more complicated functions to achieve in hardware and will take more cycles than cordic itself. This can't be avoided, but the user can try to optimize this by pipelining which will be discussed later. Once the absU is calculated, it can be fed into the same cordic which will have an accompanying x value. The same division step will have to happen when this cordic is done, which will successfully produce the required x' value. This signal is needed by every other signal in the system, that is why it is calculated first. Now that the division by An is done, three parallel divisions can start. These divisions are for cO, sRO, and sIO but they can be calculated in parallel because none of them rely on each other. Once these divisions are done the final multiplication can be done for yO, and this sequence will repeat for the next data.

This all seems to be straightforward until it is implemented. The components alone can be easily made, but once they are connected in a cascading fashion issues may appear.

B. Challenges

The initial plan for the beamformer is to have it pipelined in a way that will output valid data every clock cycle. This is due to the speed of signals, so it's always best to quickly calculate instead of wasting time. Initially the boundary cell was developed in a non-pipelined way, and then was going to be pipelined. It was later found that this cell can't be pipelined due to the calculation of x'. This signal requires the use of x and absU. The signal absU depends on the input of the system uRI and uII and can be available every clock cycle. The issue comes with x since it is always updated with the new value of x'. Since this value needs to wait for the x' to be calculated, it can't be used with the next corresponding absU. This breaks the pipeline and causes issues down the entire cell. If one step can't be pipelined, that means the rest of the cell can't be pipelined. For this reason, it was decided to complete the project without a pipeline, and try to get as little clock cycles as possible.

A second challenge came with the signal x which is internal to the cell and is always being updated. Looking at the equation, it's obvious that x reaches infinity as time goes to infinity. This is an issue with hardware because a specific fixed point format was chosen at [16 14], and it's clear that x will surpass two integer bits relatively quickly. At first it was unknown why the original developers of this beamformer method would allow such an obvious mistake. After some time was spent on it, it was clear why it must be this way. Without getting too much into the math, x going to infinity is exactly what should happen. This makes cO go to one and sO go to zero, which will cascade into the other cells and eliminate the necessary elements of the input matrix while leaving some elements untouched. This leaves some questions about whether all of this complexity needs to be included into the hardware. If x is reaching large values and uI will stay small, the new value of x won't be noticeably changed. This could mean that estimation might be able to work in substitute of actual calculation. As of now, this is just speculation and it would require extensive testing to insure the correct functioning of beamforming. If estimation would work that would mean the entire cell could be pipelined as initially planned, which would allow the beamformer to be pipelined. Instead of having around 150 cycles it could possibly be cut down to one, but that needs further research.

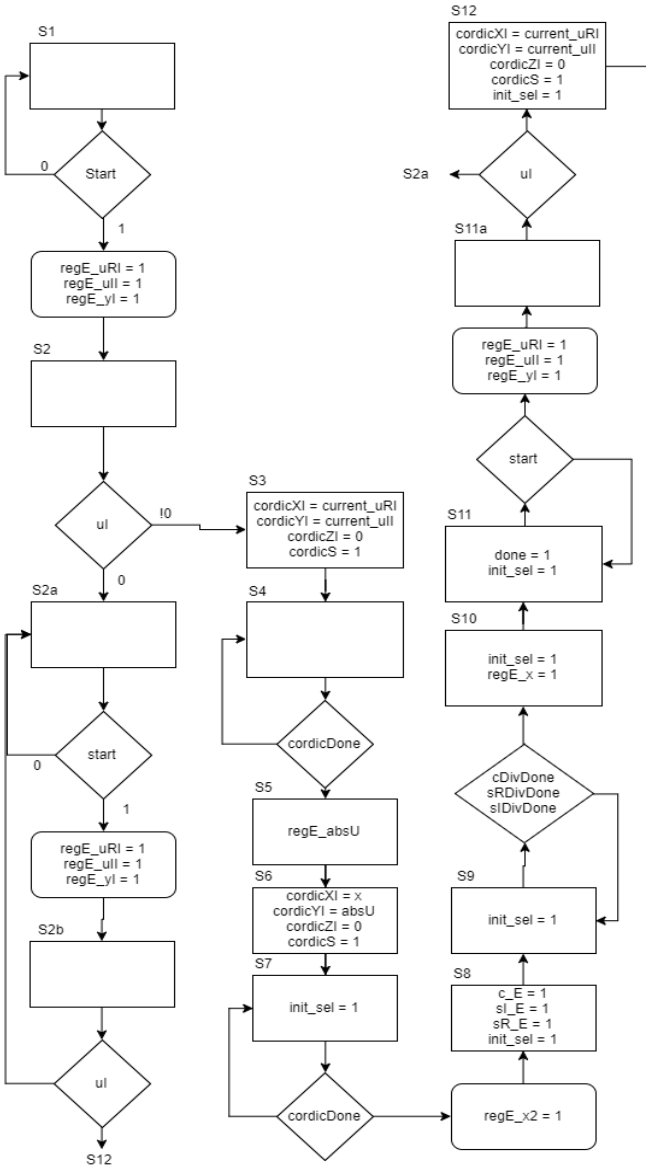

Figure 3. Boundary FSM Part 1

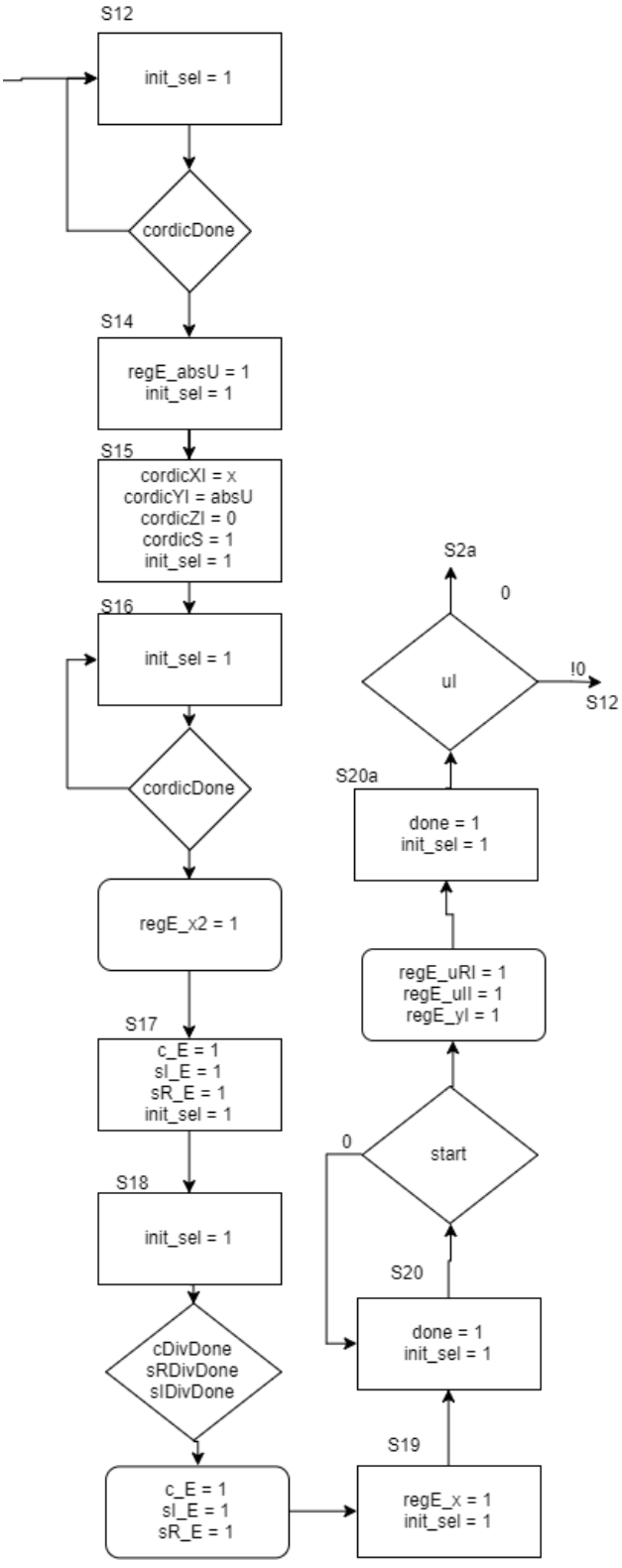

Figure 4. Boundary FSM Part 2

It can be seen in figures 3 and 4 that the FSM for the boundary cell is quite complex. This is because of an initial setup that the cell requires. There is a 99% chance that this could be simplified, but for the sake of time it was implemented the brute force way.

C. Axi-Full Protocol

With the boundary cell fully working, the next step was to design hardware that could interface between the on-board processing system and the programmable logic. This was done like previous labs in the course, which utilized an input and output FIFO which held the data that was input to the system and ready to be outputted. The details of this are extremely complex, so a high-level view of things will be discussed here. A state machine controls when data is fed into the boundary cell, started, and when the output data is ready. This is done by several signals that indicate when all of these processed should execute. The start signal goes high when enough data has been loaded to the input FIFO, and only when the done signal is high will the output FIFO be loaded. This signal will also begin the next cycle, if data is available on the input FIFO. The c code that was written to load and read data onto the FIFO's wasn't complicated, and consisted of less than 50 lines of code. Overall, including the axi-full protocol wasn't an issue.

D. 2 Antenna Beamformer

The two antenna beamformer that was also designed should be discussed here even though a functioning design hasn't been made. With the boundary cell complete, the internal cell needed to be designed and implemented.

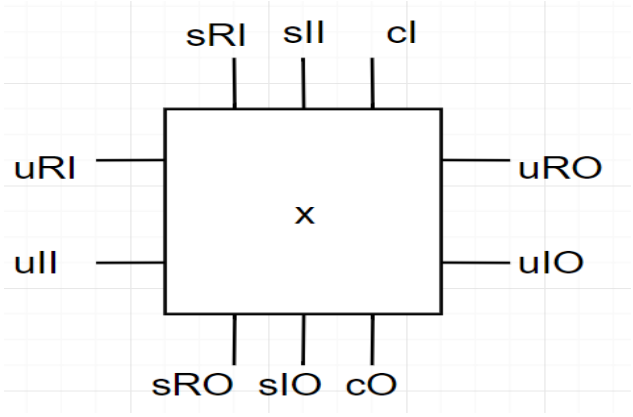

Figure 2. Internal Cell

As can be seen in Figure 2, the internal cell has many inputs and outputs like the boundary cell. The following equations determine the output values.

$$uO = c * uI - conj(sI) * x \quad [6]$$

$$x = s * uI + c * x \quad [7]$$

The challenges of this cell were mainly aimed at timing, since it's all multiplication and addition components.

Making sure the signal value is correct when it is needed was the only thing to consider in this component. Once the internal cell was complete it was time to connect it all. There were two boundary cells, three internal cells, and one final processing cell (which was one multiplication).

This was quite a challenge initially, but like the other issues with this project it was overcome after some time. The problem came with the boundary cell taking around 150 cycles to complete while the internal cell took around 20. I introduced a start and done signal to every component, so they could be in sync. This seemed to fix the problem and allowed for further development with the Axi-Full implementation. The setup for the entire beamformer was extremely like the boundary cell, except for more input signals. The state machine had additional states, but the overall process was basically identical. The only issue is that the results that were obtained from the SDK terminal didn't seem correct. There wasn't enough time to analyze the results, but there's a low chance they were right. Further research would be able to find the problem, it's most likely a small issue.

III. Experimental Setup

This project was done on a Zybo board that contains a SOC that has a Zynq-7000 FPGA and a dual core ARM processor. This allows for seamless communication between a processing system and programmable logic. To do this interaction, a protocol known as Axi-Full was used. The details of that protocol are beyond this report. To interact with the processing system, Xilinx SDK was used and communicated with the board via UART. All of this was on board with no external peripherals which made everything work smoothly. The inputs to the FPGA were mimicking signals, and the outputs were displayed to the SDK terminal.

IV. Results

The results of the boundary cell were a success. The output data that was produced on the terminal matched the expected results. This was confirmed by MATLAB and a Vivado simulation before implementing the processing system. Although the results of the beamformer weren't successful, it shouldn't be too complicated to see what went wrong. The main thing would be to go back to the simulation of the IP and see what went is happening. Once the IP is working in the simulation, the next step would be to add the AXI-Full and then simulate that. After that it would be time to test it on the board via Xilinx SDK.

Conclusions

The main take-away from this project is that theory and practice are truly different, and this needs to be considered. In theory, the beamformer should output data every clock cycle. However, as was shown above, this will be impossible without some estimation. It is impossible to pipeline a feedback loop, so there would have to be estimation for this to work as intended. Topics that were used heavily in this project were timing and pipelining. Timing is one of, if not the, most important things in hardware. Every signal needs to be its correct value when it is needed, and this isn't an easy task. The use of state machines and registers were critical in this project. Pipelined cordic, division, and multiplication were initially used until the problem of the feedback loop were found. Implementing all three was a great learning experience even if they didn't make it to the final design.

The issues that remain are the implementation of the entire beamformer. Specifically, the problems with the current rendition are unknown but with some time can be found. As stated earlier, the best place to start would be the simulation of the IP without the Axi-full hardware.

State the main take-away points from your work. List further work as well as what you learnt. What issues remain to be solved? What improvements can be made?

References

[1] Digilent, Nexys4 DDR FPGA Board Reference Manual, Rev. C, pp. 12 & 14-17, September 2014.

[2] Llamocca, D. (2017). VHDL Coding for FPGAs. Retrieved Fall 2017, from http://www.secs.oakland.edu/~llamocca/VHDLforFPGAs.html

[3] Haykin, S. (2001) *Adaptive Filter Theory 4th Edition* . Prentice Hall