VLSI Design of the Square-Root Algorithm for a Linear MMSE V-BLAST Detector with SIC

Akin Olugbade
SUPERB-CSIS 2008
Faculty Advisor: Prof. Borivoje Nikolic
Graduate Student Mentor: Vinayak Nagpal

August 1, 2008

Abstract

Traditionally, wireless communication systems use a single antenna at both the receiver and transmitter to provide means for point-to-point, single-input single-output (SISO) communications systems. Separation between different users and uses is achieved by utilizing different time and frequency domains. Recent theoretical results have shown that using the spatial domain as an additional degree of freedom in wireless communications yields higher channel capacities and better coverage by using multiple-input multiple-output (MIMO) systems. A MIMO system uses multiple antennae at the transmitter and the receiver. It can send the same signal over several antennas, which reduces the error rate or it can send and decode several independent information streams thereby increasing the rate of data flow. The focus of this research project is to study complex signal processing in a system that uses the extra spatial degrees of freedom afforded by multiple antennas to jointly decode multiple independent signals. A proposed architecture for achieving this is the Vertical Bell lab’s LAyered Space Time architecture commonly known as V-BLAST. A specific V-BLAST decoding based on the square-root algorithm has been developed using a CORDIC (COordinate Rotation Digital Computer) based implementation. The design has been simulated and implemented using a hardware description language, and is suitable for VLSI fabrication.

1 Introduction

Multiple-input multiple-output (MIMO) systems exploit the spatial domain to increase the degrees of freedom for a wireless system. This is achieved by placing multiple antennas at both the receiver and transmitter. V-BLAST (Vertical Bell Labs Layered Space Time) is an architecture proposed by Bell Labs to use the added degrees of freedom to send separate signals on each transmit antenna and jointly decode all of them with the receive antenna array [1]. This, theoretically, increases the capacity of the channel linearly with the number of transmit antennas as long as the receive array is at least as large as the transmit array. Implementing this system efficiently, however, presents many challenges. Many decoding schemes such as maximum likelihood [2] are too complex to implement and others such as zero forcing do not offer adequate performance [2]. It turns out that the square root algorithm can be implemented efficiently and yields nearly optimal performance over a wide range of SNRs. Section 2 of this paper gives background information of V-BLAST.
square root algorithm. Section 4 details implementing part of the square root algorithm using CORDIC (Coordinate Rotation Digital Computer) processing elements. Finally, section 5 offers some conclusions and further discussions.

2 Background Information

2.1 V-BLAST

V-BLAST is a MIMO architecture in which separate data streams are transmitted on M antennas. Each of these antennas transmits their signal to all N>M receiver antennas. We assume this is done over a rich scattering, flat fading channel. V-BLAST systems have been shown to have unprecedented spectral efficiencies of about 20 – 40 bps/Hz at SNRs from 20 – 34 dB. Figure 1 shows a high level diagram of V-BLAST [3].

Given our assumption of a Rayleigh flat fading we can model our system as:

\[
\mathbf{x} = \mathbf{Hs} + \mathbf{v}, \quad \mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_N \end{bmatrix}, \quad \mathbf{s} = \begin{bmatrix} \mathbf{s}_1 \\ \vdots \\ \mathbf{s}_M \end{bmatrix}
\]  

where \( \mathbf{x} \) is a complex vector of length \( N \) and represents the received symbol vector. \( \mathbf{H} \) is a complex matrix of dimensions \( N \times M \) and represents the channel matrix where an element of \( \mathbf{H} \), \( H_{ij} \sim N(0,1) \). \( \mathbf{s} \) is a complex vector of length \( M \) and represents the transmitted symbol vector. Finally, \( \mathbf{v} \) is a complex vector of length \( N \) and represents the additive white noise. \( \mathbf{s} \) has an average power of \( 1/\alpha \) and its elements are uncorrelated. \( \mathbf{v} \) too has an average power of 1 and its elements are uncorrelated. The SNR of the system is thus \( 1/\alpha \). The assumption of a Rayleigh channel tells us that the channel matrix consists of elements that are uncorrelated to one another [4].

2.2 LMMSE with SIC

The goal of the receiver is to give a good estimate, \( \hat{\mathbf{s}} \), of the transmitted symbol vector \( \mathbf{s} \). As discussed in the introduction, some schemes for decoding are too complex to implement in hardware and others do not give good enough performance. One possible scheme, however, is a Linear Minimum Mean Square Error estimate, which is given as:

\[
\hat{\mathbf{s}} = (\alpha I + \mathbf{H}^\ast \mathbf{H})^{-1} \mathbf{H}^\ast \mathbf{x} = \begin{bmatrix} \mathbf{H} \\ \sqrt{\alpha} \mathbf{I}_M \end{bmatrix}^\dagger \begin{bmatrix} \mathbf{x} \\ \mathbf{0} \end{bmatrix} \quad (2)
\]

The error covariance matrix for this estimate is:

\[
E(\hat{\mathbf{s}} - \mathbf{s})(\hat{\mathbf{s}} - \mathbf{s})^\ast = (\alpha I + \mathbf{H}^\ast \mathbf{H})^{-1} \equiv \mathbf{P} \quad (3)
\]

In this error covariance matrix, \( \mathbf{P} \), the strongest received information stream corresponds to the element \( P_{11} \) that is smallest. We can then rearrange the elements of \( \hat{\mathbf{s}} \) such that the strongest signal is the Mth element, decode this strongest signal, and then null out the Mth element from the received vector \( \mathbf{x} \). We can then recalculate \( \hat{\mathbf{s}} \) and \( \mathbf{P} \) using a deflated version of \( \mathbf{x} \) and \( \mathbf{H} \). We repeat until we have done this for every element in \( \hat{\mathbf{s}} \) and decoded every signal. This process is known as Successive Interference Cancellation (SIC) and has been shown to enable transmission at close to capacity for a wide SNR range [5]. This is shown in Figure 2.
The main problem with SIC is that it requires repeated recalculation of $P$ using operations that are expensive to do in hardware. A proposed alternative to this approach is the Square Root Algorithm. In [4] Babak Hassibi shows that instead of calculating the error covariance matrix, $P$, we can calculate the Square Root of this matrix $P^{1/2}$. In addition to $P^{1/2}$ being easier to calculate, the smallest element of $P^{1/2}$ still corresponds to the strongest signal, and once $P^{1/2}$ has been calculated it turns out that there is no need to recalculate it for the following steps of SIC (for details refer to [4]). The remainder of this paper focuses on the VLSI design of components that find $P^{1/2}$ given $H$ and $x$.

2.3 The Square Root Algorithm

The algorithm for finding $P^{1/2}$ is as follows:

\[
\begin{bmatrix}
1 & H_{i}^{1/2} \\
0 & P_{i-1}^{1/2}
\end{bmatrix}
\begin{bmatrix}
\theta_i \\
e_i
\end{bmatrix} =
\begin{bmatrix}
r_{e,i}^{1/2} & 0 \\
-K_i & P_{i-1}^{1/2}
\end{bmatrix}
\begin{bmatrix}
A_i \\
Q_i
\end{bmatrix}
\]

(4)

where $H_i$ is the ith row of $H$ and $P_i^{1/2} = \frac{1}{\sqrt{a}} I_M$ and $Q_0 = 0_{N \times M}$. $e_i$ is an $N \times 1$ vector with all elements 0 except the $i$th element which is 1. $\theta_i$ represents some unitary transformation. The general idea of the algorithm is to do some unitary transformation on the pre-matrix such that we zero out the row $H_{i}^{1/2}$ to obtain $P_{i-1}^{1/2}$. This $P_{i}^{1/2}$ is then used in the next step as $P_{i-1}^{1/2}$. This process is repeated $N$ times until we have:

\[P^{1/2} = P_{0}^{1/2}\]

(5)

2.4 Givens Rotations

A unitary transformation used to zero elements in a column is the Givens Rotation. The basic equation for a Givens Rotation is:

\[
\begin{bmatrix}
\cos \theta & \sin \theta \\
-\sin \theta & \cos \theta
\end{bmatrix}
\begin{bmatrix}
x \\
y
\end{bmatrix} =
\begin{bmatrix}
x' \\
y'
\end{bmatrix}
\]

(6)

where $\theta$ is picked such that $y' = 0$. This rotation can be used on a larger matrix

\[
\begin{bmatrix}
1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\
\vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\
0 & \cdots & c & \cdots & s & \cdots & 0 \\
\vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\
0 & \cdots & -s & \cdots & c & \cdots & 0 \\
\vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\
0 & \cdots & 0 & \cdots & 0 & \cdots & 1
\end{bmatrix}
= 
\begin{bmatrix}
x \\
\vdots \\
y
\vdots \\
y'
\vdots \\
\vdots
\end{bmatrix}
\]

(7)

This transformation only affects the value to be made zero and the element with which it was rotated - its pivot. The Givens matrix is the identity matrix with the corresponding elements replaced. We use this transformation in equation (4) to zero out the necessary row. We use the first column of the matrix in (4) as a pivot. We do this for each row, in order not to change the determinant of the matrix. We zero our desired element and rotate its corresponding values in each row by using a pivot in the corresponding row of the first column. Thus the first rotation with our “lead” element determines the angle, and the successive rotations with “following” elements are of the same angle as our leader.

A complex Givens rotation can be performed by rotating a two element complex vector in the complex plane such that its imaginary part becomes zero. This is done using the same idea as a normal Givens rotation in which $x$ and $y$ correspond to a complex number’s real and imaginary part. After the complex number has been made real, a normal Givens rotation can be done. More details can be found in [6].

2.5 CORDIC

In hardware, an efficient way of accomplishing a Givens rotation is using a CORDIC. A CORDIC implements the rotation equations

\[
x' = \cos \theta (x - y \tan \theta) \\
y' = \cos \theta (y + x \tan \theta)
\]

(8)

when angles are selected such that

\[
\tan \theta = 2^{-i}
\]

(9)

In this case, multiplication by $\tan \theta$ simply becomes a right shift. When several of these CORDIC processing elements are used together,
one can rotate by an arbitrary angle by rotating by a combination of allowed angles:

$$\theta = \tan^{-1} 2^{-i} \quad (10)$$

For a rotation using a fixed number of iterations the $\cos \theta$ terms turn out to be a constant. The constant scaling value can be seen in [6] for up to 15 iterations. For our design we need the CORDIC to first rotate a vector to the nulling axis and then remember the angle rotated to following vectors can be rotated to the same angle. These two modes of operation are known as vectoring and rotation, respectively.

3 Design

3.1 Design Platform

The main design platform used during this project was SIMULINK. It is a tool-flow that enables the use and creation of high level block diagrams which can be used for simulation, emulation, and hardware description. The blocks used in this design were from the Xilinx block set and blocks developed by members of the Berkeley Wireless Research Center. The designs were made for an FPGA platform called the Berkeley Emulation Engine 2 (BEE2). The BEE2 is made up of five Virtex –II Pro FPGAs.

3.2 Design of CORDIC

The design of our CORDIC implemented the rotation equations (8) using the constraint on angles in (10) such that our final result nulls $y'$. We also needed to design a CORDIC that operates in vectoring and rotation mode.

In order to implement the equations, we used shifters and adders to do the bulk of the work along with simple decision logic. Each processing element receives two input vectors and finds their sign. It must now decide based on their signs whether to rotate up or down. We found that a simple XOR function works in determining this. We also perform the multiplication by $\tan \theta$ shifting by $i$ bits in the $i$th processing element. Next we use this decision to rotate up or down. This corresponds to either setting or not setting the subtract bit for our adder/subtractors.

In order to enable vectoring and rotating modes in our design, we have an additional input. When it is set the decision logic is saved in a register and also used in that iteration. When it is not set, the decision logic for rotating up or down is read from the register. Since the angle by which the vector is rotated during each iteration is fixed, it is identical in both rotation and vectoring mode. A full block diagram of the CORDIC processing element implemented in this project can be seen in Figure 3.

![Figure 3: A block diagram of a CORDIC processing element.](image1)

We then designed the logic to connect these CORDIC processing elements in series to attain rotation close to the nulling axis. The logic is fully pipelined, i.e. reads in a new vector every clock cycle. This is accomplished by Figure 4 shows an example of a chain of pipelined CORDIC processing elements for six stages. This is accomplished by storing values in registers every clock cycle as well as propagating the vectoring mode signal throughout the chain of CORDIC processing elements.

![Figure 4: This shows an example of the view underneath the main CORDIC block. It shows a pipelined chain of six CORDIC processing elements.](image2)
The last stage of the CORDIC is a simple constant multiplier. It can be shown that for a fixed number \( i \) of processing elements, multiplying the \( \cos \theta \) terms corresponding to every stage turns out to be a constant [6]. This block scales the CORDIC output by the correct constant which changes with the number of stages used.

3.3 Simulink Mask Scripting

Every block designed was designed to be parameterized. Details about the block can be changed and the block will automatically adjust to reflect these changes. This parameterization was accomplished using Simulink Mask Scripting. It is a largely undocumented technique and thus took some time to learn and understand well enough for proper use. However, once understood, mask scripting turned out to be a very powerful tool for creating reusable and flexible library blocks.

The CORDIC sub block which contained the chain of CORDIC processing elements was the first to use mask scripting. It was vital to its design. One can set the number of processing elements to be used and thus determine the accuracy to which vector will be rotated. Simulink mask scripting was used to dynamically redraw this chain in accordance with the parameter. Masks scripting was also used to propagate information down to the processing elements in order set the correct shift amount used for its iteration.

Mask scripting was also used to set the constant by which the results would be scaled. Since the number of processing iterations determines this constant, it was necessary to use Simulink mask scripting to determine the value to use and update for parameter for the multiplication block.

Another useful application of mask scripting was a borrowed technique used by Xilinx on all of their blocks. The block has printed on itself its latency. This also is calculated and updated when relevant parameters are changed. This can be seen in Figure 4. An example of parameters that can be changed is given in Figure 5.

Figure 5: An example dialog box showing editable parameters.

3.4 Finding \( P_{1/2} \)

Using this CORDIC, we finally designed what is called the “Super Cell” in [7]. This is the block that can find \( P_{1/2} \). It consists of three full CORDIC blocks. The first CORDIC is called the theta CORDIC. The theta CORDIC reads in a column from the matrix in (4). The leading element of this column needs to be made zero. The column is rotated in complex plain in the theta CORDIC to make the leading element real. We then use two more CORDICS called the master phi CORDIC and the slave phi CORDIC. The first column of the matrix in (4) is used as the pivot column to rotate the output of the theta CORDIC such that the leading real output of the theta CORDIC is rotated to be zero. The master phi CORDIC rotates the leading real output of the theta CORDIC with the real part of the pivot column to make it zero. This angle also dictates the angle by which imaginary parts of the theta stage are rotated with the corresponding imaginary parts of the pivot column in the slave phi CORDIC. The outputs of the Super Cell are used to update the respective
columns of the matrix. This is repeated N times and the final result gives us $P^{1/2}$ as described in (4). This process is visualized in Figure 6.

3.5 Results
Every block was tested extensively in simulation. Testing was done by performing the algorithm for that block in MATLAB to obtain the expected values given certain data. The blocks were then given the same inputs as the algorithm was given in MATLAB, simulation was run, and the outputs of the blocks were reviewed. All of the blocks performed as desired, given several known test inputs of a wide range.

4 Conclusion
While the full Square Root algorithm was not designed, the major computationally complex parts were. Finding $P^{1/2}$ enables one to easily perform SIC and subsequently decode information streams in V-BLAST architecture. V-BLAST is not currently used in any current military or commercial wireless standards. The research done in this project will be used in ongoing research at the Berkeley Wireless Research Center and help further research in the area of V-BLAST detection. With the potential data rate gains offered by V-BLAST, it may very well see uses in products in the future.

![Diagram](image)

**Figure 6:** The row given by $H_1P^{1/2}_{i-1}$ is to be nulled. First we perform a complex Givens rotation on column being operated on, and then we use the first column of the matrix as a pivot to zero the element using a regular givens rotation.
References

Acknowledgements
The author would like to thank the contributions of the students, faculty, and sponsors of the Berkeley Wireless Research Center, the National Science Foundation Infrastructure Grant No. 0403427. Vinayak Nagpal and Borivoje Nikolic are also gratefully acknowledged for the time and guidance they both have given.