

Reconfigurable Processing Framework for Space-Time Block Codes

S Kuo, I V McLoughlin and K Mehrotra

Group Research
Tait Electronics Ltd
P O. Box 1645
Christchurch, NZ

shyh hao.kuo@tait.co.nz; ian.mcloughlin@tait.co.nz; kishore.mehrotra@tait.co.nz

Abstract – Space-time block coding has emerged in recent years as a promising research topic to enable future high-speed wireless communications networks and services. A considerable number of authors have developed algorithms and techniques that have been explored in simulations. However, fully implemented systems are rare. This paper considers the use of reconfigurable logic for implementing space-time block coding algorithms, allowing greater coding complexity than traditional DSP implemented systems. In particular, a number of matrix multiplication-style architectures are proposed for implementing channel estimators on FPGA hardware. These are compared in terms of speed and size, and accuracy is compared with a MATLAB™ simulation.

I. INTRODUCTION

Space time (ST) techniques are an active research area for future high speed or high capacity wireless communication systems. A number of ST techniques have been demonstrated such as the Alamouti scheme [1], time reversal (TR) space time block coding (STBC) [2][3] and others. Working systems typically involve substantial processing requirements that may limit their adoption in commercial implementations. Estimates of the processing requirement for a 2-channel Alamouti implementation are around 3 billion multiply-accumulate calculations per second [4].

Traditional digital processing implementations typically involve the use of digital signal processors (DSP), specialised microprocessors running at core speeds of up to about 720MHz at the current time. Leading edge DSPs may contain dual multiply-accumulate (MAC) cores which, when fully utilised, would yield peak processing performance of approximately 5700 MMACS (million 16-bit multiply-accumulate operations per second) [4][5].

In recent years, reprogrammable logic in the form of field programmable gate arrays (FPGAs) has emerged as a serious contender for signal processing implementation [6]. Although FPGA implementation may be less rapid than DSP implementation, FPGAs may allow greater aggregate processing capability.

For example, the Stratix EP1S80 FPGA from Altera contains 80,000 logic elements and 80 dedicated 9-bit embedded multipliers [7]. The embedded multipliers can each operate at up to 300MHz and be paired to allow 16-

bit operation (similar to typical DSP multipliers). This alone provides up to 12,000 MMACS. In addition, a proportion of the 80,000 spare logic elements, each of which contain a look-up-table and flip-flop, could also be used to implement MAC functions. Since one discrete multiplier requires less than 400 logic elements, up to 200 MAC functions can be provided at 300MHz, providing another 60,000 MMACS. In reality, the entire FPGA cannot be utilised for processing, and implementations are rarely this efficient. However, it is evident that, in terms of raw processing power, the FPGA provides greater performance than a DSP.

This paper describes the use of an FPGA for implementing a STBC communications scheme that operates in real-time. In this instance, the scheme has a modulation rate of 2Mbits/s using a five-fold oversampling TR-STBC engine over a 2.4GHz carrier, where transmit data is bridged between two ethernet networks.

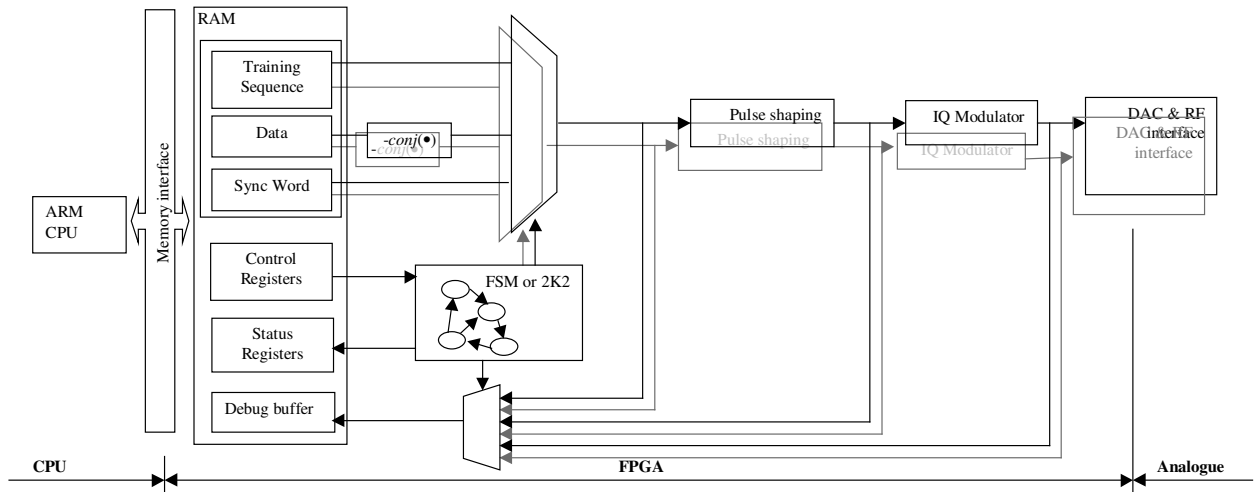
The implementation size as a proportion of available resource is measured in terms of occupied logic elements on an Altera EP1S25 (having 80 embedded multipliers and 25,000 uncommitted logic elements).

II. SYSTEM DESIGN

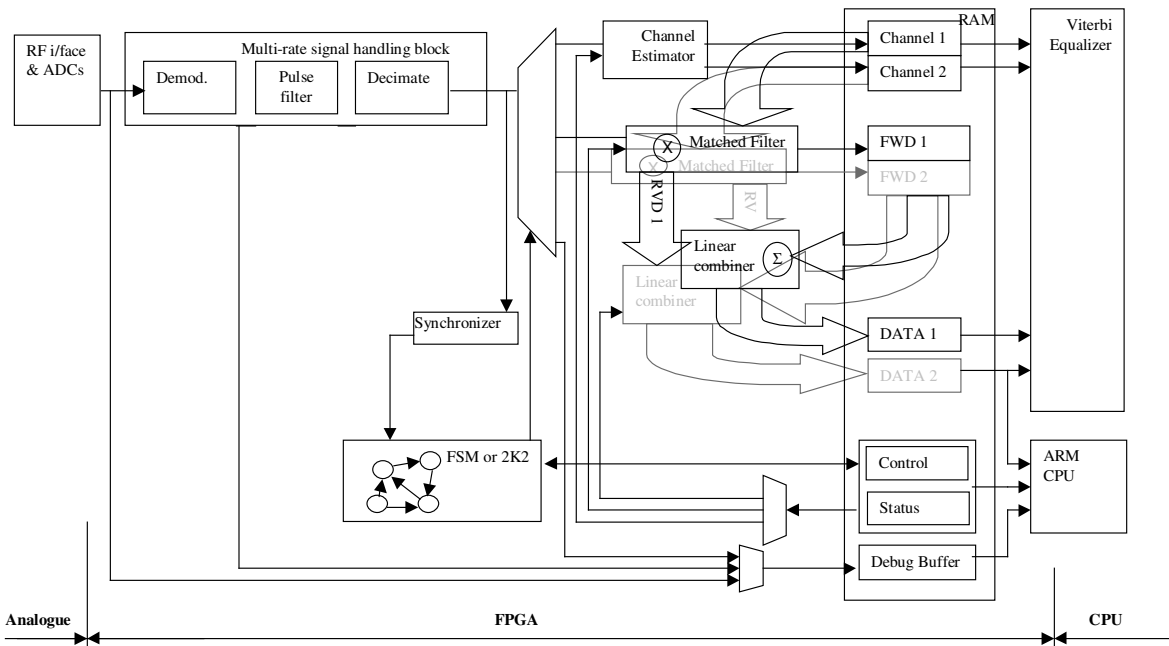
The hardware implementation of the communication link utilising a TR-STBC code is a significant leap in digital signal processing technology. Although the theory is well understood and can be simulated with relative ease, it still poses a significant challenge when it comes to digital hardware implementation.

The system digital architecture for both the transmitter and receiver is shown in *Figure 1*. The architecture reflects the processing stages required and, at the same time, shows each hardware sub-module that forms a complete work package to be developed and implemented independently of each other.

The signal processing hardware platform consists of a CPU, a digital logic stage and an analogue stage. For the TR-STBC configuration, most of the processing is implemented in digital logic to take advantage of the speed and parallel processing capability of the FPGA. The CPU is only used for user interface and debugging purposes.



(a)



(b)

Figure 1 – Digital architecture of the TR-STBC transmitter (a) and receiver (b).

A. Transmitter signal path

The digital logic of the transmitter comprises a large block of memory for storing the data payload as well as fixed sequences such as training and synchronisation words. A finite state machine (or a proprietary embedded controller) is used to switch the appropriate sequence into the transmit path according to the predefined packet structure. Note that the diagram shows a second grey image of the transmit path in the background for the second transmit antenna to show the modules that need to be duplicated for 2 channel TR-STBC system.

B. Receiver signal path

The TR-STBC receiver has only one receive antenna. The received signal is firstly demodulated and decimated to a reasonable sample rate. In this instance, the sample rate is

10M samples per sec. A circuit for detecting the synchronisation sequence is used on the decimated signal to provide frame timing to the finite state machine to provide a trigger for the rest of the processing circuit.

A least squares estimate is used to calculate the channel impulse response based upon the received training sequence. The TR-STBC payload is then fed through the TR-STBC decoder to separate the two transmitted signals. The two separated signals are then passed through Viterbi equalizers to finally extract the payloads.

III. RE-CONFIGURABLE PROCESSING FRAMEWORK

Current state-of-the-art reconfigurable digital hardware such as the Stratix FPGA allows quick prototyping of large digital processing hardware structures. This paper describes a framework consisting of VHDL components and design methodology that allows quick prototyping of

space time processing structures. The aim of the framework is to allow a real-time prototype to be build without the usual problem associated with debugging real-time digital hardware and software.

As an example, this paper presents in detail the design and verification for a generic matrix multiplier block which is used for channel estimation in the current configuration.

A. Matrix multiplication as a generic building block

Most receiver processing blocks or operations can be written in the form of

$$\mathbf{y} = \mathbf{A}\mathbf{x}, \quad (1)$$

where \mathbf{x} and \mathbf{y} are vectors depicting input and output time sequences and \mathbf{A} is a linear transformation that performs a signal processing function such as block code decoding, channel estimation, equalisation, or even discrete Fourier transform.

Earlier work on matrix computations has mainly been done on optimising memory access and memory sharing structures for a single or a small number of general-purpose processors.[8][9]. With the recent development of large programmable logic devices[6], the optimisation criteria has been shifted towards using many (10 to 100) but more specialised processing unit such as a multiplier and accumulators.

The matrix multiplier assumes that the input vector \mathbf{x} is presented to the multiplier block one element at a time. This is true for most receiver processing blocks where the input comes directly or indirectly from the analogue to digital converter at the front end. A quick examination of the mechanism of the matrix multiplication reveals that any element of \mathbf{y} depends on all of the elements in \mathbf{x} . The most obvious implementation of the matrix multiplication is to buffer all of the inputs and allow some processing time after receiving the last x to calculate the individual elements of \mathbf{y} using the expansion shown in equation (2).

$$\begin{aligned} y_1 &= A_{1,1}x_1 + A_{1,2}x_2 + \dots + A_{1,n}x_n \\ y_2 &= A_{2,1}x_1 + A_{2,2}x_2 + \dots + A_{2,n}x_n \\ &\vdots \\ y_m &= A_{m,1}x_1 + A_{m,2}x_2 + \dots + A_{m,n}x_n \end{aligned} \quad (2)$$

The output format can be optimised for parallel output with low latency at the expense of requiring m identical circuits for calculating y_1 to y_m (*Figure 2*); or formatted serially to reuse the same circuit but at the expense of a higher latency (*Figure 3*).

B. Pipelining for speed

The matrix multiplier is a very versatile block that may be used many times within a single receiver configuration. Therefore, it may be desirable to optimise each individual

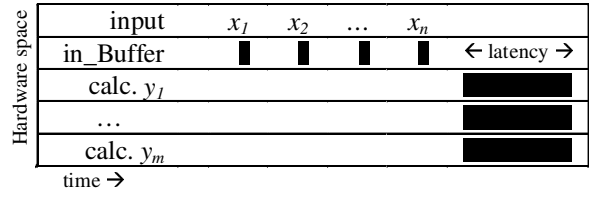


Figure 2 - Process scheduling for a matrix multiplier with parallel output.

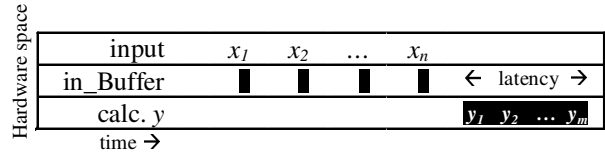


Figure 3 - Process scheduling for a matrix multiplier with serial output.

instance of the matrix multiplier to take advantage of any structure in the \mathbf{A} matrix. Many optimisation techniques exist for Toeplitz, Hermitian, sparse and other special matrices. However, it is a time consuming and error prone exercise to tailor every instance of the matrix multipliers in a given configuration when the aim is to build a research prototype to test a concept.

The approach taken in the design of the matrix multiplier specifically avoids optimisation that relies on the structure of the matrix \mathbf{A} . Instead, we exploit the time sequencing of the input vector. Mathematically, the matrix multiplication is expanded as shown in equation (3).

$$\begin{aligned} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} &= \begin{pmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,n} \\ A_{2,1} & A_{2,2} & \dots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m,1} & A_{m,2} & \dots & A_{m,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \\ &= \begin{pmatrix} A_{1,1} \\ A_{2,1} \\ \vdots \\ A_{m,1} \end{pmatrix} x_1 + \begin{pmatrix} A_{1,2} \\ A_{2,2} \\ \vdots \\ A_{m,2} \end{pmatrix} x_2 + \dots + \begin{pmatrix} A_{1,n} \\ A_{2,n} \\ \vdots \\ A_{m,n} \end{pmatrix} x_n \end{aligned} \quad (3)$$

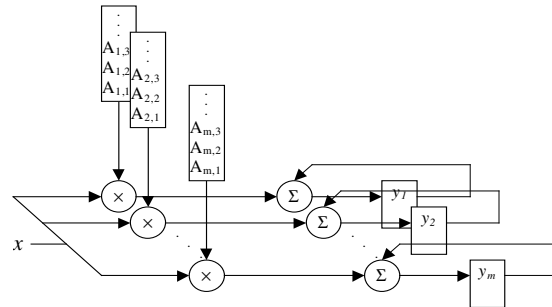


Figure 4 – Hardware structure that performs serial Matrix multiplication.

In this form, the matrix multiplier is factored into n partial sums each depending only on one element of \mathbf{x} . This factoring allows the elementary multiplications to start as soon as x_l is received, and thus eliminates the buffering latency. The hardware structure that implements a matrix multiplier following equation (3) is shown in *Figure 4*.

This structure allows the processing load to be distributed across the entire time duration of receiving x as shown in the process scheduling (*Figure 5*). Another advantage of this structure is that there is no longer the need to provide storage space for the input buffer.

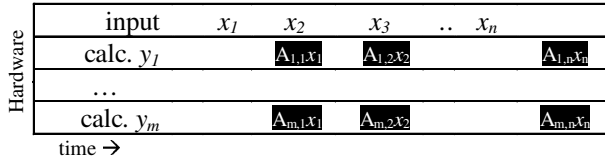


Figure 5 - Process scheduling for a serial matrix multiplier with parallel output.

There is a two-dimensional trade-off between the fully parallel and the fully serial implementations. The partial serial implementation chosen lies at a pragmatic minimum between these where the choice of parallelism and pipeline depth is chosen structurally based on matrix size rather than on tailored implementation cost.

C. Channel Estimation Example

The matrix multiplier structure described previously has been used in the channel estimation module of the TR-STBC communication system to perform least squares channel estimation. The module is written in VHDL with fully parameterisable size and coefficients to allow late changes in the training sequence.

Further optimisation and pipelining has been performed to fully utilise the resources provided by the specific programmable device used. This results in a matrix multiplier that has the output ready for the next stage of processing within one symbol period of receiving the last element in \mathbf{x} .

The function of the channel estimation module is to calculate least squares estimates of the channel impulse response \mathbf{H} as

$$\hat{\mathbf{H}} = \begin{bmatrix} \hat{\mathbf{H}}_1 \\ \hat{\mathbf{H}}_2 \end{bmatrix} = (\mathbf{S}^H \mathbf{S})^{-1} \mathbf{S}^H \mathbf{Y} \quad (4)$$

where \mathbf{Y} is the received training sequence and \mathbf{S} is the Toeplitz convolution matrix constructed from the training sequence.[4]

To aid the development of the channel estimation module, MATLAB scripts were created to automate the process of converting a matrix into the memory map to be used by the matrix multiplier. This allows us to automate the complete test cycle as shown in *Figure 6*.

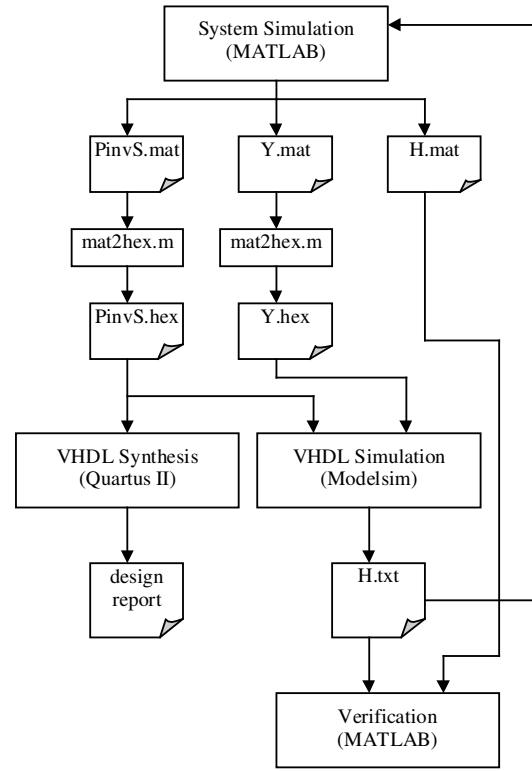


Figure 6 – Simulation to working module process flow chart.

In the simplest form, the system simulation generates random inputs for calculating equation (4) and stores the pseudo-inverse, input and output vectors in hex files as shown. The VHDL simulation can then take the input files and perform the matrix multiplication as described previously. The output can be dumped into a text file for comparison. A typical output from this stage is shown in *Figure 7*. This allows quick visual verification of the functionality of the channel estimator.

Similarly, the system simulation could be used to search for an optimum training sequence and convert the output to hex format for programming the final system without the need of editing VHDL directly.

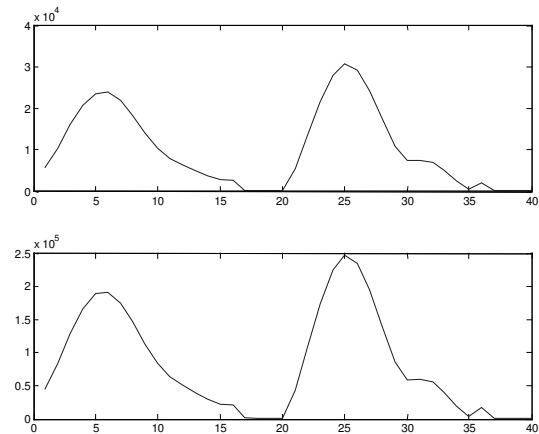


Figure 7 – A typical output from the verification stage showing the magnitude (y-axis) of the 40 tap channel estimates(x-axis) from MATLAB(top) and VHDL(bottom)

Lastly, for a system simulation of the complete transmit and receive chain, the output from the VHDL simulation can be read back into the MATLAB simulation to provide insight into quantisation effect and noise performance.

IV. ANALYSIS

A. Size and Speed trade-off

The processing load for calculating a general matrix multiplication contains the same number of multiplications and additions irrespective of the method chosen. Assume we have a matrix multiplication between a matrix of size m by n and a vector of size n by 1. A full size matrix multiplier where all of the multiplications and additions are to be performed simultaneously requires $m.n$ multipliers and $m(n-1)$ adders in addition to $n+m$ registers for the input and output buffer.

By performing the multiplication and addition in series, it is possible to re-use the same multiplier and adder multiple times and if a latency corresponding to $m.n$ operations can be tolerated. the circuit can be reduced down to 1 multiplier, 1 adder and $n+m$ registers for the input and output buffer.

The proposed structure with the serial input vector has the optimal latency of the parallel circuit and at the same time, only requires n multipliers, n adders and n registers.

B. Performance consideration for FPGA

The fundamental building block of an FPGA is the logic element (LE) which consists of a look-up table (LUT) followed by a register [10]. The LUT is used to perform any combinational logic such as add, and multiply while the register is used for storing the output of the LUT for the next stage. Due to the tight coupling between the LUT and the register, the number of LEs used in a design is constrained by the larger of the two. For the three matrix multiplier examples used before, the estimated number of logic elements needed for each design is shown in **Table 1**.

Implementation	parallel	serial	optimised
multipliers used (N_{multi})	nm	1	m
adders used (N_{add})	$nm-m$	1	m
registers used (N_{reg})	$n+m$	$n+m$	m
LE estimated $\max(N_{multi}, N_{add}) + N_{reg}$	$nm+n+m$	$n+m+1$	$2m$

Table 1 – size comparison between three different matrix multiplier implementations.

The actual implementation of the optimised design consists of 40 taps of 32-bit numbers which correspond to a estimated design size of 2560 logic elements. The actual number of LEs used in the final implementation is 2829 LEs comprising 2560 LEs in the processing core and 269 LEs for the control logic.

V. CONCLUSION

FPGA logic provides a flexible and powerful platform for implementing communications signal processing functions. For Space-Time block coding implementation which requires high level of computational resources, FPGA may be preferable to the DSP-implemented algorithms. This paper presents a streamlined matrix multiplication sub-block, the channel estimator for a typical STBC algorithm that is implemented on FPGA. Two traditional implementation strategies were presented: fully parallel and fully serial, in addition to a more pragmatic optimal solution. Using the approach described, new signal processing algorithms can be readily converted from simulation into fully implemented digital hardware systems.

VI. REFERENCES

- [1] S. M. Alamouti, "A simple transmit diversity technique for wireless communications," *IEEE J. Select. Areas Commun.*, vol 16, pp.1451-1458, Oct. 1998
- [2] E. Linskog, A Paulraj, "A Transmit Diversity Scheme for Channels with Intersymbol Interference", *In Proceedings of ICC'2000*, pp 307-311, June 2000
- [3] P. Stoica, E. Linskog, "Space Time Block Coding for Channels with Intersymbol Interference", *In proceedings of 35th Asilomar Conference on Signals Systems and computers*. pp 252-256, 2001
- [4] K. Mehrotra, I. V. Mcloughlin, "Time Reversal Space Time Block Coding with Channel Estimation and Synchronisation errors", Submitted for publication in ATNAC 2003
- [5] Texas Instruments SPRU189F, "TMS320C6000 CPU and Instruction Set Reference Guide," (Rev. F) [online] Available: <http://focus.ti.com/lit/ug/spru189f/spru189f.pdf>
- [6] R. Hartenstein, H. Grünbacher (Editors), *The Roadmap to Reconfigurable computing*, *In proceedings of FPL2000*; LNCS, Springer-Verlag 2000
- [7] Altera white paper, "Stratix Device Background," Feb 2002, ver. 1.0, [online] Available: http://www.altera.com/literature/wp/wp_stx_backgro under.pdf
- [8] G H Golub, C F Van Loan, *Matrix Computations*, 3rd ed. The Johns Hopkins University Press, 1996
- [9] E. Anderson, et. al, *LAPACK Users' Guide*, Release 2, 2nd ed. SIAM Publications, 1995
- [10] Altera Document, "Stratix Device Family Datasheet," Jan 2003, ver. 3.0, [online] Available: http://www.altera.com/literature/ds/ds_sgx.pdf