UCLA
UCLA Electronic Theses and Dissertations

Title
MIMO Accelerator: Programmable MIMO Decoder Chip and Design Environment

Permalink
https://escholarship.org/uc/item/0x34m19n

Author
Mohamed, Mohamed Ismail Ali

Publication Date
2012

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA

Los Angeles

MIMO Accelerator:

Programmable MIMO Decoder Chip and Design Environment

A dissertation submitted in partial satisfaction of the

requirements for the degree Doctor of Philosophy

in Electrical Engineering

by

Mohamed Ismail Ali Mohamed

2012

ABSTRACT OF THE DISSERTATION

MIMO Accelerator:

Programmable MIMO Decoder Chip and Design Environment

by

Mohamed Ismail Ali Mohamed

Doctor of Philosophy in Electrical Engineering

University of California, Los Angeles 2012

Professor Babak Daneshrad, Chair

With wireless communications becoming an essential part of human life, wireless technology advances to meet the increasing demands. New standards are introduced every couple of years to regulate the implementation of wireless systems. Most of modern standards are based on MIMO and OFDM signaling, which makes any time saving in a MIMO-OFDM receiver design cycle essential and the support of multi-standards in the same device highly desirable.

This work introduces a hardware implementation for a MIMO decoder accelerator, which is a software-programmable device that specializes in MIMO decoding, and MIMO signal processing in general, for OFDM systems. A VLSI implementation of the accelerator is introduced highlighting some of the implementation decisions and techniques to minimize the overall energy consumption of the accelerator hardware. The accelerator chip core area is $2.48mm^2$ in 65nm CMOS technology. Its average power consumption is 224.3 at 166MHz clock

frequency. A deeply pipelined design for a powerful processing core allows the accelerator to achieve energy consumption figures competing with specialized designs. A single accelerator chip can be programmed to complete 4x4 QR decomposition, 4x4 Singular-Value Decomposition (SVD), 2x2 MMSE MIMO decoding, 4x4 MMSE MIMO decoding, or many other possible applications.

A simple design flow is presented to assist a MIMO-accelerator user in mapping a MIMO-related algorithm to a successful accelerator-based hardware implementation in no time. The accelerator, with its diversity and energy efficiency, can empower a wireless MIMO-OFDM receiver giving it an unparalleled advantage over regular fixed-data-path systems.

The Dissertation of Mohamed Ismail Ali Mohamed is approved.

Danijela Cabric

William Kaiser

Milos Ercegovac

Babak Daneshrad, Committee Chair

University of California, Los Angeles

2012

To my dear wife, Tasnim

And to the victims of the great Egyptian revolution of Jan 25th 2011

which I missed while producing this work

Table of Contents

1. INTRODUCTION **1**

1.1. MIMO AND OFDM OPERATION 4

1.1.1. MIMO and Spatial Multiplexing *4*

1.1.2. OFDM for Wide Band Signaling *6*

1.2. CHANNEL EQUALIZATION FOR MIMO-OFDM SYSTEMS 8

1.2.1. Linear MIMO Decoders *10*

1.2.2. Maximum-Likelihood MIMO Decoders *12*

1.2.3. Singular-Value Decomposition for MIMO Decoding *15*

1.2.4. Iterative MIMO Decoders *16*

1.3. OUTLINE 17

2. MIMO ACCELERATOR HARDWARE: OVERVIEW **19**

2.1. TOP VIEW OF THE ACCELERATOR 19

2.2. PROCESSING CORE 21

2.2.1. Addition and Subtraction Unit *22*

2.2.2. Reciprocal Unit *23*

2.2.3. Multiplication Unit *24*

2.2.4. Rotation Unit *25*

2.3. DYNAMIC SCALING 30

2.4. MEMORY SWITCHES 30

3. MIMO ACCELERATOR HARDWARE: IMPLEMENTATION CHALLENGES AND TRADEOFFS **33**

3.1. INSTRUCTION FETCH AND PIPELINE 33

3.2. CONTROLLER COUNTERS PIPELINE 37

3.3. DATA MEMORY STRUCTURE 39

3.4. PROCESSING CORE INPUT-GATING 43

3.5. SYSTEM INTEGRATION 44

3.6. MULTI-ALGORITHM SWITCHING 48

4. MIMO ACCELERATOR DESIGN FLOWS AND PROGRAMMING 50

4.1. HARDWARE PARAMETERS 50

4.2. MIMO ACCELERATOR DESIGN FLOWS 51

4.2.1. Hardware-First Design flow 51

4.2.2. Algorithm-First Design Flow 53

4.3. MIMO ACCELERATOR PROGRAMMING 55

4.3.1. Bit-Level Instruction 55

4.3.2. High-Level Instructions 57

4.3.3. Compiler Interface 65

4.3.4. Example Program 67

4.4. HARDWARE INSTANCE GENERATION 68

4.5. PARAMETER EXTRACTION 70

5. MIMO ACCELERATOR PROTOTYPE CHIP 72

5.1. TEST SETUP AND PROCEDURE 73

5.2. MEASUREMENT RESULTS 78

6. CONCLUSIONS 87

7. FUTURE WORK 88

REFERENCES 89

List of Figures

Fig. 1.1. Our goal compared to programmable hardware (such as DSP) and dedicated ASICs 3

Fig. 1.2. Channel coefficients for M transmitters and N receivers 5

Fig. 1.3. Each OFDM subchannel effectively suffers only flat fading 7

Fig. 1.4. A block diagram for a 2x2 MIMO-OFDM system 9

Fig. 2.1. MIMO Accelerator block diagram 20

Fig. 2.2. Block diagram for the addition/subtraction processing unit for four receive streams 22

Fig. 2.3. Block diagram for the reciprocal processing unit for four receive streams 23

Fig. 2.4. Block diagram for the multiplication processing unit for four receive streams 24

Fig. 2.5. Block diagram of the rotation processing-unit for four receive streams. 26

Fig. 2.6. Complex vectoring CORDIC 27

Fig. 2.7. Complex rotation CORDIC 27

Fig. 2.8. Phase Processing (PP) block diagram as a part of the rotation unit. 28

Fig. 2.9. A rerouted version of the complex vectoring CORDIC 29

Fig. 2.10. Rerouted versions of the complex rotation CORDIC. 29

Fig. 2.11. Memory switches: (a) Core-Input Switch, Right, (b) Memory-Input Switch 31

Fig. 3.1. Accelerator instruction pipeline with four stages 34

Fig. 3.2. Accelerator pre-decoded instruction structure 35

Fig. 3.3. Accelerator instruction pipeline with five stages after introducing the extra fetch stage 37

Fig. 3.4. The two instances of the controller: read controller (left) and write controller (right) 38

Fig. 3.5. (a) Linear data memory structure, (b) One register for the data memory 39

Fig. 3.6. Proposed data memory structure (for N_M=4) 40

Fig. 3.7. Implemented sectored data memory (for N_M=4) 42

Fig. 3.8. Processing core after gating the four processing units 43

Fig. 3.9. Overloading one memory port to allow top-level access to the data memory contents 45

Fig. 3.10. Multiplexing the two data memory ports between regular operation and top-level access 47

FIG. 3.11. TOP-LEVEL CONTROL OVER THE RUNNING PROGRAM THROUGH THE ADDRESS OF THE INSTRUCTION MEMORY............49

FIG. 4.1. HARDWARE-FIRST DESIGN FLOW TO UTILIZE THE MIMO ACCELERATOR PROGRAMMABILITY 52

FIG. 4.2. ALGORITHM-FIRST DESIGN FLOW TO OPTIMIZE THE MIMO ACCELERATOR HARDWARE FOR A SPECIFIC APPLICATION .54

FIG. 4.3. ACCELERATOR PRE-DECODED INSTRUCTION STRUCTURE 56

FIG. 4.4. MIMO ACCELERATOR COMPILER GRAPHICAL USER INTERFACE...66

FIG. 4.5. MIMO ACCELERATOR INSTANCE GENERATION GRAPHICAL USER INTERFACE 69

FIG. 4.6. PARAMETER EXTRACTION GRAPHICAL USER INTERFACE...71

FIG. 5.1. TEST SETUP FOR THE MIMO ACCELERATOR CHIP..73

FIG. 5.2. PHOTO FOR THE LAB TEST SETUP IN 74

FIG. 5.3. ON-CHIP TEST HARDWARE BLOCKS..75

FIG. 5.4. ON-CHIP PARALLEL-TO-SERIAL AND SERIAL-TO-PARALLEL CONVERTERS 76

FIG. 5.5. TIMING DIAGRAM FOR THE INTERACTION BETWEEN THE TEST SIGNALS TO CONTROL THE CLOCK MULTIPLEXING 78

FIG. 5.6. MIMO ACCELERATOR CHIP MICROGRAPH...78

FIG. 5.7: POWER CONSUMPTION AND ENERGY PER ONE CLOCK CYCLE VERSUS CLOCK FREQUENCY, ...82

AND LEAKAGE POWER VERSUS SUPPLY VOLTAGE 82

FIG. 5.9. BER COMPARISON BETWEEN MMSE ON THE ACCELERATOR AND FLOATING POINT SIMULATION 83

FIG. 5.9. PER COMPARISON BETWEEN MMSE ON THE ACCELERATOR AND FLOATING POINT SIMULATION 84

List of Tables

TABLE 4.1. MIMO ACCELERATOR HDL PARAMETERS AND THEIR VALUES FOR THE ASIC PROTOTYPE....51

TABLE 4.2. THE STRUCTURE OF A MIMO ACCELERATOR BIT-LEVEL INSTRUCTION....56

TABLE 4.3. STYLES FOR THE RESULT AND THE OPERANDS IN A HIGH-LEVEL INSTRUCTION....58

TABLE 4.4. MULTIPLICATION, ADDITION, AND RECIPROCAL OPERATORS IN THE ACCELERATOR INSTRUCTION SET....59

TABLE 4.5. BRANCHING OPERATORS IN THE MIMO ACCELERATOR INSTRUCTION SET....60

TABLE 4.6. ROTATION OPERATORS IN THE MIMO ACCELERATOR INSTRUCTION SET....61

TABLE 4.7. SPECIAL NOP INSTRUCTIONS....64

TABLE 4.8. EXAMPLE PROGRAM USED FOR 2X2 MMSE MIMO DECODING....67

TABLE 5.1. MIMO ACCELERATOR HDL PARAMETERS AND THEIR VALUES FOR THE ASIC PROTOTYPE....72

TABLE 5.2. THE PERCENTAGE OF THE MAIN BLOCKS AREA TO THE COMPLETE CHIP CORE AREA....79

TABLE 5.3. TIME NEEDED TO FINISH AN ALGORITHM RUNNING ON THE ACCELERATOR PROTOTYPE....80

TABLE 5.4. SUMMARY FOR A SUBSET OF RELEVANT 802.11N AND LTE-A PARAMETERS....81

TABLE 5.5. COMPARISON TO OTHER ASIC DESIGNS....85

1. Introduction

In a wireless transceiver design, three major steps form a typical process of translating a digital-signal processing task into hardware implementation. First is to choose and design the algorithm that will be used to achieve this task. This step usually includes a high level simulation, whether it is a floating-point simulation or a fixed-point simulation or both, to check if this particular algorithm satisfies the performance constraints, such as bit error rate (BER) or packet error rate (PER). Second is the design of a digital circuit that implements the chosen algorithm. The circuit design takes into account all the hardware requirements in terms of area (cost), speed (throughput and latency), and power consumption. The digital circuit design is usually based on a hardware description language (HDL) such as Verilog and VHDL. A test bench in HDL is written to perform a function test based on test vectors for the circuit inputs and its expected outputs. These test vectors are usually generated from the simulation model of the first design step. Third is combining this particular circuit with different ones for the physical implementation of the complete transceiver (or part of it) into a chip. The physical implementation means – in abbreviation – to synthesize the HDL into a standard-cells library for an application-specific integrated circuit (ASIC) implementation or into logic cells for a field-programmable gate array (FPGA), and then perform placement and routing for the final chip. This physical implementation process usually contains some kind of iteration based on the results of post-synthesis and post-layout simulations or based on formal verification.

Despite the fact that this design process is time consuming, it has to be repeated for the same transceiver component if the system requirements change. The requirements usually change with the release of new wireless communication standards, which occurs as frequent as every year.

Reducing the consumed time in a repeated design cycle is essential for cost and time-to-market reduction. Time saving can be achieved by attacking the third design step, the physical design, by using a FPGA instead of an ASIC as the final implementation platform. This might be possible if the hardware requirements allow the speed degradation and the boost in power consumption that comes with a FPGA. But the unit cost for a FPGA solution will be much higher than an ASIC solution for a mass production.

Design time saving can also be achieved by attacking the digital-circuit design step. One approach is to use High-Level Synthesis (HLS). In HLS, the algorithm is written in C, C++, or SystemC and a HLS tool translates it directly to HDL. The time saving comes from the fact that writing the algorithm in a high-level language like C++ is simpler than implementing the digital circuit from scratch. The drawback of this solution is that the outcome is highly dependent on the used HLS tool and whether it can produce a hardware that is as optimized as a hand-written HDL can be. And to reach a better result from a HLS tool, the designer has to closely control the HLS process to guide the tool to the required results. While the HLS technology is appealing, it will take more time for it to be the industry standard.

Another approach to reduce the time consumed in circuit design is to use a programmable solution such as a Digital-Signal Processor (DSP). Using a DSP converts the circuit design task into a much easier problem of translating the algorithm into an efficient software program for the DSP in hand. The drawback of this approach is the expected degradation in performance and power consumption due to the use of a general-purpose processor to solve a specific problem.

In some cases, the programmability of the final design may be the drive factor for using a DSP. The saving in design time makes a programmable solution very appealing, but adds up to this is the increasing demand for multi-standard devices. Devices such as smart phones and tablets, for example, support WiFi and LTE. Both standards are based on Orthogonal Frequency Division Multiplexing (OFDM) and, currently, Multiple-Input Multiple-Output (MIMO) operation. If a programmable hardware is used, it can support the two standards with the same piece of hardware.

Based on the mentioned motivations, the goal of this work is to present the *MIMO accelerator* as a programmable solution for the MIMO decoding problem for OFDM systems without sacrificing the performance. As shown in Fig. 1.1, this work will sacrifice some re-usability by limiting our target to MIMO decoding for OFDM systems instead of general purpose DSP design. But we aim to gain a huge performance improvement that will take this programmable device to a performance that is very close to dedicated ASIC designs. The performance metric that we use is the energy consumption as it is the limiting factor of hardware performance on mobile devices.

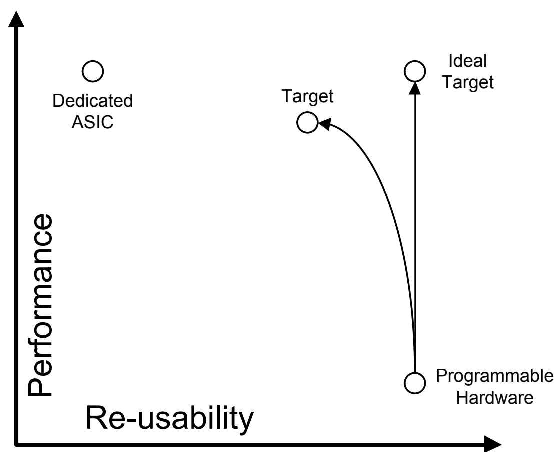

Fig. 1.1. Our goal compared to programmable hardware (such as DSP) and dedicated ASICs

With a complete framework for MIMO decoder implementation, the MIMO accelerator almost completely eliminates the effort and time for the hardware circuit design without compromising the hardware requirements. The MIMO accelerator framework builds upon a complex-vectors-based processor that targets the MIMO decoding problem. Before diving into the MIMO accelerator details, a brief introduction for MIMO and OFDM operation is presented.

1.1. MIMO and OFDM Operation

Most, if not all, of the recently released (and the upcoming in the near future) wireless-communication standards are based on MIMO-OFDM operation. This applies to small-scale modern wireless data networks such as the 802.11 wireless LAN standard [1], and also applies to large-scale cellular systems such as WiMAX [2] and LTE [3] (and its enhanced version LTE-A [4]) that are considered 4G candidates. As this work focuses on MIMO decoders for OFDM systems, this section will give a brief introduction to both MIMO and OFDM operation.

1.1.1. MIMO and Spatial Multiplexing

A MIMO system uses multiple antennas at both ends of the system – the transmitter and the receiver. For a narrow-band system, Fig. 1.2 shows a simple diagram for a transmitter with *M* antennas and a receiver with *N* antennas. Each antenna at the receiver side receives signals from all the M transmitting antennas with different channel coefficients. This is called an MxN MIMO system and can be represented in a matrix format as

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} h_{11} & \cdots & h_{1M} \\ h_{21} & & h_{2M} \\ \vdots & \ddots & \vdots \\ h_{N1} & \cdots & h_{NM} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_M \end{bmatrix} + \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_N \end{bmatrix} \qquad (1.1)$$

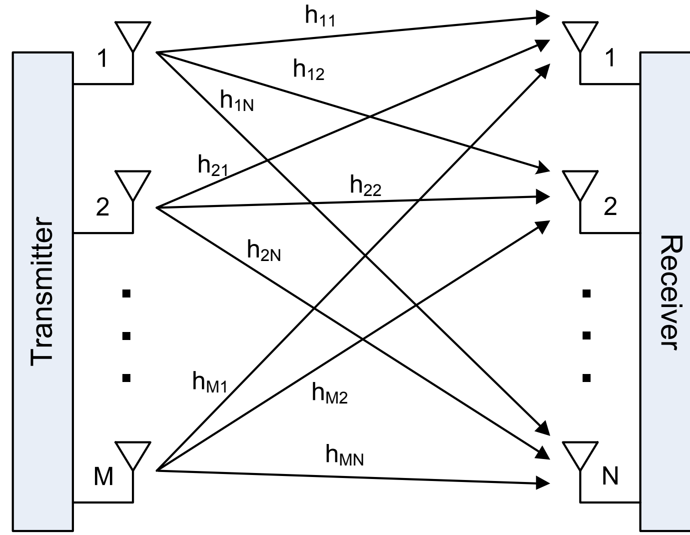

Fig. 1.2. Channel coefficients for M transmitters and N receivers

Where y_1 to y_N are the received symbols at the N receive antennas and x_1 to x_M are the transmitted symbols from the M transmit antennas. h_{nm} is the channel coefficient from the transmit antenna *m* to the receive antenna *n*. And z_1 to z_N are the AWGN. For simplification, equation (1.1) can be rewritten as

$$\boldsymbol{y} = H.\boldsymbol{x} + \boldsymbol{z} \qquad (1.2)$$

Where $\boldsymbol{y}$ is the received vector, $\boldsymbol{x}$ is the transmitted vector, $\boldsymbol{z}$ is the noise vector, and H is the channel matrix.

Based on the channel characteristics and the receiver signal-to-noise ratio (SNR), a transmitter can send a different data stream per antenna to increase the system throughput (capacity), or it can use the antennas for redundancy to increase the system reliability. In a high scattering environment, a city for example, a channel can support more independent paths. This is reflected in the rank of the channel matrix *H*; a higher rank means more independent paths. In such a case, the transmitter will transmit M data streams over the M transmit antennas in the same time slot

and frequency band, which is called *spatial multiplexing* given the fact that the antennas are scattered in space. For spatial multiplexing without any added diversity, the receiver needs a number of antennas (N) to be the same as the number of spatial data streams leading to an NxN system.

If the channel status doesn't allow multiple data streams, a MIMO system can be used to introduce diversity. A single data stream can be transmitted over a 1xN system to introduce receiver diversity. Techniques such as selection combining, threshold combining, and maximal-ratio combining can be used to increase the effective receiver SNR for better reliability [5]. A single data stream can also be transmitted over an Mx1 system to introduce transmitter diversity by applying techniques such as Alamouti scheme or space-time block codes (STBC) in general [6].

In general, a system doesn't have to be limited to either use spatial multiplexing or spatial diversity. A combination of the two may be used to maximize the benefit from the available hardware. Adaptive systems can change from an antenna configuration to another or from introducing diversity to multiplexing based on channel status and variation [7]. In this dissertation, a spatial multiplexing system – with the same number of antennas at the two sides – is always assumed except if mentioned otherwise.

1.1.2. OFDM for Wide Band Signaling

In a system that uses a wide bandwidth for higher throughputs, transmitting the data symbols over a single carrier suffers from the frequency selectivity of the channel. Sophisticated techniques have to be used for channel equalization at the receiver to avoid inter-symbol

interference (ISI). An OFDM system avoids this problem by dividing a wide channel into a number of narrow sub-channels in a hardware-efficient manner.

From frequency selectivity point of view, transmitting multiple subcarriers (a subcarrier per sub-channel) transforms the wide-band fading problem to several narrow-band fading problems, as shown in Fig. 1.3. Narrow-band fading channel equalization can be as easy as dividing the received symbols by a channel coefficient. From another perspective, the symbol time of a wide band signal is very small (the inverse of the bandwidth) compared to the channel delay spread T_m, which causes a severe ISI. By using several subcarriers, the symbol time is multiplied by the number of subcarriers, which decreases the effect of the channel delay spread and the ISI. The small ratio of T_m to the symbol time allows OFDM systems to introduce guard intervals to eliminate the effect of the ISI without a big loss in throughput. A cyclic prefix is used in this guard interval for phase continuity, reducing the linear convolution to a cyclic-convolution for discrete systems [5].

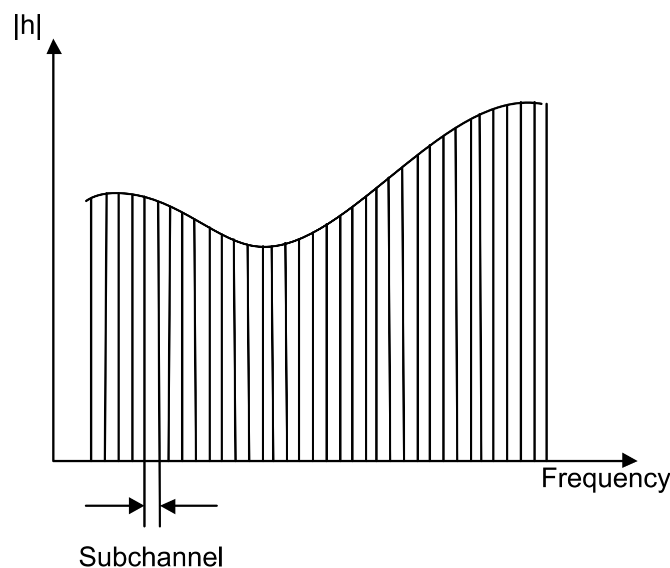

Fig. 1.3. Each OFDM subchannel effectively suffers only flat fading

1.2. Channel Equalization for MIMO-OFDM Systems

For an OFDM system, each sub-channel faces flat fading leading to a relatively easy channel equalization task. But this comes with the cost of parallel processing requirements that result from dividing the frequency band into a number of sub-channels. If an OFDM system uses N_{sc} subcarriers, the flat-fading channel equalization has to be repeated N_{sc} times. And for MIMO-OFDM, each subcarrier carries a MIMO signal. MIMO signaling transforms the equalization problem formulation and solution into complex matrix operations. For a spatial-multiplexing MIMO system, with N_{rx} receive antennas, the complexity of an equalizer increases exponentially with N_{rx}. These extensive computation requirements for both OFDM and MIMO signaling raise the challenges for the hardware implementation of MIMO-OFDM channel equalizers.

Fig. 1.4 shows a simplified block diagram for a 2x2 MIMO-OFDM transceiver. In the transmitter side, the data bit stream is scrambled, encoded, interleaved, and then divided into the MIMO spatial streams. As shown in Fig. 1.4, many elements throughout the transceiver are repeated for the MIMO operation. The Inverse Fast Fourier Transform (IFFT) is used at the transmitter side to construct the time-domain OFDM signal in a hardware efficient manner. The inverse operation (FFT) is used on the receiver side to reconstruct the frequency-domain OFDM signal. In addition to the reverse operations of the transmitter, a MIMO-OFDM receiver performs extra operations for timing-synchronization – such as packet and symbol timing detection – and extra operations for frequency synchronization – such as the frequency offset estimation and correction.

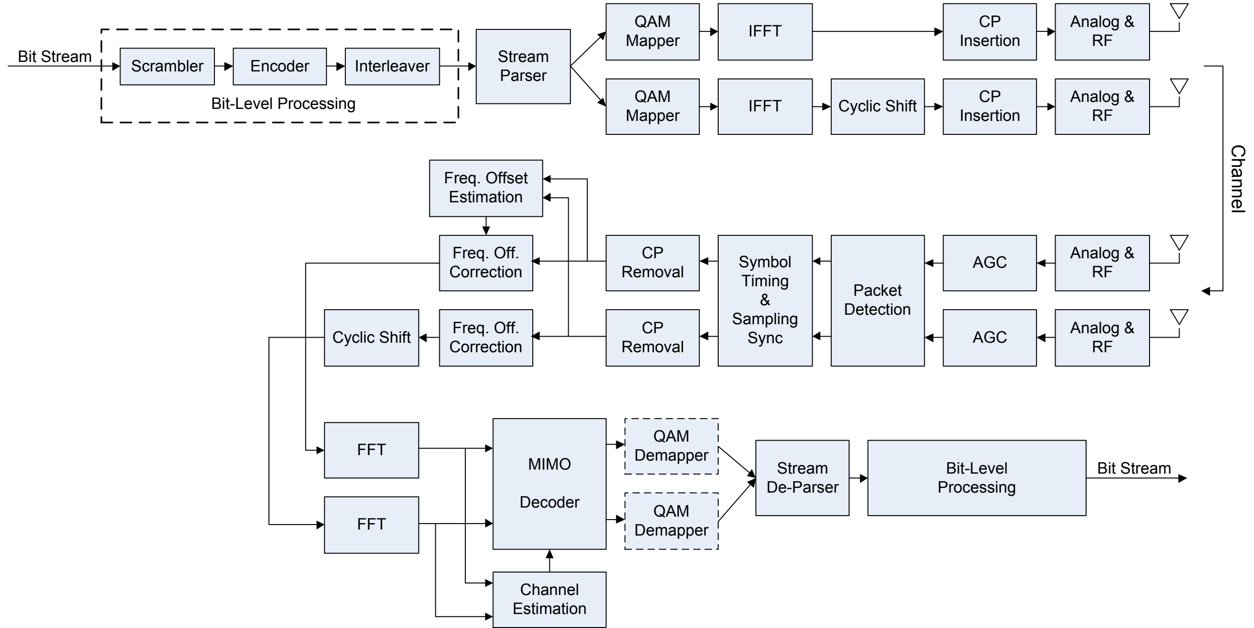

Fig. 1.4. A block diagram for a 2x2 MIMO-OFDM system

The receiver block of interest in Fig. 1.4 is the MIMO decoder. A MIMO decoder carries the task of channel equalization for a MIMO system. A receiver uses a preamble and pilots in the system data frame structure to estimate the channel coefficients [1-3]; a coefficient is estimated for every transmit-receive antenna pair to form a channel matrix. This is repeated for each subcarrier. A MIMO decoder then takes these channel estimates and uses them to estimate the transmitted data symbols using the received data symbols.

To describe the MIMO decoder operation and various algorithms, equation (1.3) repeats the model of equation (1.2) but for a MIMO-OFDM system. If $\boldsymbol{y}_i[n]$ is the received data vector at the MIMO decoder input for a subcarrier index *i* and OFDM symbol with time-index *n*, then

$$\boldsymbol{y}_i[n] = H_i.\boldsymbol{x}_i[n] + \boldsymbol{z}_i[n] \qquad (1.3)$$

$\boldsymbol{y}_i[n]$ is a complex vector of size $N_{rx}\times 1$, where N_{rx} is the number of receive streams (antennas). H_i is the $N_{rx}\times N_{rx}$ estimated channel matrix for subcarrier *i*, assuming the same number of

antennas on both sides of the channel (spatial multiplexing). $\boldsymbol{x}_i[n]$ is the $N_{rx}\times 1$ transmitted complex data vector for subcarrier i and OFDM symbol n. And $z_i[n]$ is the AWGN vector of size $N_{rx}\times 1$. A MIMO decoder task is to estimate the transmitted data vector $\boldsymbol{x}_i[n]$ for every subcarrier i and OFDM symbol n.

1.2.1. Linear MIMO Decoders

MIMO decoding algorithms can be divided into two groups: linear and non-linear MIMO decoders. A linear MIMO decoder is based on the computation of the inverse of the channel matrix H – or a variant of it. The received symbols are then multiplied by the inversion result, W, as shown in equation (1.4), omitting the time and subcarrier indexes for brevity.

$$\hat{\boldsymbol{x}} = W\boldsymbol{y} = WH\boldsymbol{x} + W\boldsymbol{z} \qquad (1.4)$$

Where $\hat{\boldsymbol{x}}$ is an estimation of the transmitted vector $\boldsymbol{x}$.

The direct implementation of a linear MIMO decoder is a Zero Forcing (ZF) decoder. In ZF, the W matrix is the inverse of the channel matrix ($W = H^{-1}$). The pseudo-inverse of the channel matrix, equation (1.5), is used to guarantee an inverse.

$$W_{ZF} = (H^H H)^{-1} H^H \qquad (1.5)$$

Where H^H indicates the Hermitian transpose (complex-conjugate transpose) of the channel matrix H.

The ZF algorithm suffers from noise enhancement in highly faded channels [8][9]. The channel inverse in such cases boosts the amplitude (power) of the noise vector $\boldsymbol{z}$, which leads to a SNR reduction that directly affects the BER and PER of the system. To avoid this problem, the

Minimum Mean-Square Error (MMSE) linear MIMO decoder takes the noise variance into account for computing the W matrix [8][5], equation (1.6).

$$W_{MMSE} = (H^H H + N_0 I)^{-1} H^H \qquad (1.6)$$

Where N_0 is the noise variance and I is the identity matrix.

The hardware implementation of a linear MIMO decoder – whether it is a ZF or a MMSE decoder – mainly requires matrix-matrix multiplication, vector-matrix multiplication, and matrix inversion. In the literature, hardware implementation of MMSE focuses on efficiently implementing matrix inversion by relaxing the hardware requirements in terms of area and throughput – such as [10], [11], [12], [13], and [14]. The hardware implementation in [10] is based on the Sherman-Morrison formula [15] as a special case of the matrix inversion lemma. While in [11], [12], [13], and [14] the hardware implementation is based on a QR factorization of the matrix-to-be-inverted.

A QR factorization of a matrix A ($A = QR$) represents the matrix as a multiplication of a unitary matrix Q and an upper-triangular matrix R. A unitary matrix is a matrix whose inverse is the Hermitian transpose of the matrix ($Q^H Q = I$). And the matrix inversion of an upper-triangular matrix is simple compared to a regular matrix [11]. By performing the QR factorization for the matrix to-be-inverted in W_{ZF} or W_{MMSE}, the hardware requirements will be reduced to computing the matrix inverse of a triangular matrix as shown in equation (1.7).

$$A^{-1} = (QR)^{-1} = R^{-1} Q^{-1} = R^{-1} Q^H \qquad (1.7)$$

In [16], a matrix-inversion-free implementation of the MMSE is reported based on a higher order QR factorization. Based on the notions in equation (1.6) for W_{MMSE}, and assuming the channel matrix H to be of size $N_{rx}\times N_{rx}$, a matrix G can be formed and QR-factorized such as

$$G_{2N_{rx}\times N_{rx}} = \begin{bmatrix} H_{N_{rx}\times N_{rx}} \\ \sqrt{N_0} I_{N_{rx}\times N_{rx}} \end{bmatrix} = Q_{2N_{rx}\times N_{rx}} R_{N_{rx}\times N_{rx}} = \begin{bmatrix} Q_{1,N_{rx}\times N_{rx}} \\ Q_{2,N_{rx}\times N_{rx}} \end{bmatrix} R_{N_{rx}\times N_{rx}} \qquad (1.8)$$

Where Q is a unitary matrix and both Q_2 and R are upper-triangular matrices. From this $2N_{rx}\times N_{rx}$ QR factorization, the channel matrix can be rewritten as

$$H = Q_1 R \quad , \quad H^H = R^H Q_1^H \qquad (1.9)$$

And

$$R^{-1} = \frac{1}{\sqrt{N_0}} Q_2 \qquad (1.10)$$

Based on (1.9), (1.10), and the fact that Q is a unitary matrix, W_{MMSE} can be rewritten as

$$W_{MMSE} = (H^H H + N_0 I)^{-1} H^H = \frac{1}{\sqrt{N_0}} Q_2 Q_1^H \qquad (1.11)$$

Using this technique for a MMSE MIMO decoder reduces the hardware requirements to a single QR factorization, a matrix multiplication, and a reciprocal calculation.

1.2.2. Maximum-Likelihood MIMO Decoders

In a linear MIMO decoder, an explicit inverse of the channel matrix, or a variant of it, is calculated as an initial step. The inversion result is then multiplied by the received vector in a linear equation to eliminate the channel effect. Non-linear MIMO decoders follow a different route. Maximum-Likelihood (ML), as a non-linear MIMO decoder, follows a more intuitive way. In a ML decoder, an exhaustive search in all possible vectors is done to find the best candidate to be considered the received vector. This search detects the minimum Euclidean distance between

the received vector and each possible vector if it is to be received, as shown in the following equation.

$$\hat{\boldsymbol{x}} = arg \min_{\tilde{\boldsymbol{x}} \epsilon P^{N_{rx}}} ||\boldsymbol{y} - H\tilde{\boldsymbol{x}}||^2 \qquad (1.12)$$

Where P is a pool of all constellation points and N_{rx} is the number of receive streams.

The complexity of a direct implementation of equation (1.12) in a ML MIMO decoder exponentially increases with both the number of constellation points and the number of received streams, N_{rx} [17]. And for an OFDM system, this calculation has to be repeated for N_{sc} subcarriers. To avoid the exhaustive search of a ML decoder, more efficient search methods, such as Sphere Decoder (SD), is used.

In a SD, the received vector $\boldsymbol{y}$ is considered a point in a space with N_{rx} dimensions. A candidate vector in the pool of possible vectors, or points, is excluded if its distance from $\boldsymbol{y}$ is greater than a radius d of a hypothetical sphere that spans all the N_{rx} dimensions. The choice of this radius d is a design problem; it may depend on the SNR for example. A point survives if it passes the test of equation (1.13).

$$||\boldsymbol{y} - H\hat{\boldsymbol{x}}||^2 < d^2 \qquad (1.13)$$

Where $\hat{\boldsymbol{x}}$ is a candidate point, which is a vector with N_{rx} elements and each elements is a constellation point.

To reduce the complexity from an exhaustive search, SD starts with a QR factorization for the channel matrix H.

$$\|\boldsymbol{y} - QR\hat{\boldsymbol{x}}\|^2 < d^2 \qquad (1.14)$$

Given the fact that a unitary transformation doesn't affect a vector's norm, equation (1.14) can be modified to be

$$\|Q^H\boldsymbol{y} - R\hat{\boldsymbol{x}}\|^2 < d^2 \qquad (1.15)$$

$$\|\boldsymbol{l} - R\hat{\boldsymbol{x}}\|^2 < d^2 \quad , \quad \boldsymbol{l} = Q^H\boldsymbol{y} \qquad (1.16)$$

Then based on the triangular matrix R, equation (1.16) can be expanded to

$$\sum_{i=1}^{N_{rx}} \left(l_i - \sum_{j=i}^{N_{rx}} r_{i,j}\hat{x}_j \right)^2 < d^2$$

$$\sum_{i=1}^{N_{rx}} p_i^2 < d^2 \quad , \quad p_i^2 = \left(l_i - \sum_{j=i}^{N_{rx}} r_{i,j}\hat{x}_j \right)^2 \qquad (1.17)$$

Where l_i, $r_{i,j}$, and $\hat{x}_j$ are elements of $\boldsymbol{l}$, R, and $\hat{\boldsymbol{x}}$ respectively.

Equation (1.17) is the base for the SD reduction in the candidate search. If only one term, p_i^2, of the N_{rx} terms of the outer summation of equation (1.17) dissatisfies the condition, then the whole candidate vector $\hat{\boldsymbol{x}}$ dissatisfies the condition. A search can then be considered a tree that starts with the simpler calculation of $p_{N_{rx}}^2$. If for a particular $\hat{\boldsymbol{x}}$ the value of $p_{N_{rx}}^2$ came out to be greater than d^2, then all candidates that share the N_{rx} th element of this particular $\hat{\boldsymbol{x}}$ are excluded from the candidate search. For the remaining candidates, on the first level of the tree, $p_{N_{rx}}^2 + p_{N_{rx}-1}^2$ is calculated. When this calculation exceeds the limit, all candidates that share the last two elements of the candidate-under-test are excluded. This continues until all N_{rx} levels of the tree are visited. More about this formulation of the SD and the tree explanation can be found in [17].

A SD, as described, can be used to reach the maximum likelihood solution of MIMO decoding. But its hardware implementation usually suffers from an unfixed throughput, which might generate problems when integrated in a bigger system. Many papers for SD hardware implementation have been published to tackle the hardware complexity and throughput challenges such as [18][19]. Some SD designs and variants reduce the hardware complexity on the expense of not guaranteeing ML detection, such designs are usually called near-ML or close-to-ML sphere decoders [20].

1.2.3. Singular-Value Decomposition for MIMO Decoding

In a MIMO system, each receive-antenna gets a version of the data that is transmitted by each and every transmit-antenna. The channel effect on those data paths forms the channel matrix H, as discussed. The complexity of the MIMO decoder rises from the matrix operations, or the exponentially increasing exhaustive search, that is needed to estimate the received data. Using a Singular-Value Decomposition (SVD) on the channel matrix is a mean to decouple the data streams on the receiver side. This decoupling transforms the problem of MIMO decoding N_{rx} correlated data streams to decoding N_{rx} Single-Input Single-Output (SISO) data streams [5].

The SVD of the channel matrix is a factorization of the channel matrix H into two unitary matrices, U and V, and a diagonal matrix Σ.

$$H = U\Sigma V^H \qquad (1.18)$$

In the literature, as shown in equation (1.18), the unitary matrix V is usually stated in its Hermitian transpose form for a reason that will be clear shortly.

If the transmitter has channel state information, it multiplies the transmit vector $\boldsymbol{x}$ by the unitary matrix V, as shown in equation (1.19)

$$\overline{\boldsymbol{x}} = V\boldsymbol{x} \qquad (1.19)$$

Then the received signal can be written as

$$\boldsymbol{y} = H\overline{\boldsymbol{x}} = U\Sigma V^H V\boldsymbol{x} = U\Sigma\boldsymbol{x} \qquad (1.20)$$

And given the fact that U is a unitary matrix, the receiver calculates

$$\overline{\boldsymbol{y}} = U^H\boldsymbol{y} = U^H U\Sigma\boldsymbol{x} = \Sigma\boldsymbol{x} \qquad (1.21)$$

This way, each element in the $\overline{\boldsymbol{y}}$ vector depends on one element in the $\boldsymbol{x}$ vector, as Σ is a diagonal matrix. The receiver can deal with the MIMO signal as a group of SISO signals.

The hardware complexity of this way of decoding the MIMO signal is in the implementation of the SVD itself [21][22]. There is also the necessity of giving the transmitter information about the channel state. The channel information is usually available on the receiver side, and it adds to the latency of the system to supply the transmitter by this information as well.

The SVD can also be used to simplify the matrix inversion of linear MIMO decoder. The inverse of the unitary matrices (U and V) is just the Hermitian transpose. And the inverse of the diagonal matrix is to invert the diagonal elements.

1.2.4. Iterative MIMO Decoders

One of the methods that can be used for MIMO decoding is to perform the decoding task iteratively. An iterative decoder starts to decode one of the N_{rx} receive streams by considering

the other $N_{rx} - 1$ signal as noise or interference. After decoding that first signal, the receiver subtracts its effect from the rest of the receive streams. The decoder then repeats the operation over the remaining $N_{rx} - 1$ streams. An example for iterative decoding is the Vertical Bell-labs Layered Space-Time (V-BLAST) architecture [23][24].

Despite the simplicity of the algorithm, iterative decoding suffers from two drawbacks. First is the error propagation. An error in decoding the first stream affects the decoding of the remaining streams. To reduce the effect of error propagation in V-BLAST, the decoder starts the iterations by the stream with the highest SNR. Second drawback is the latency in the system due to the need of completely decoding a stream before starting the next. The iterative decoding is simplified by starting with QR decomposition [25]. In V-BLAST, this QR is sorted to start with the highest SNR, as mentioned before.

1.3. Outline

Based on this brief introduction about our goal and motivations, and after going through a description of MIMO, OFDM, and MIMO decoding algorithms, the next chapter (chapter 2) gives an overview of the MIMO accelerator hardware, and how it can be used for MIMO decoding of OFDM systems.

Chapter 3 goes through some of the hardware implementation challenges and tradeoffs. It will show how the processing core, memory access, and system integration are all optimized for a better performance in terms of throughput and energy consumption.

Chapter 4 presents the software part of the MIMO accelerator. It describes the hardware parameters and how it can be used and configured. It also describes the accelerator programming language and tool flow.

Chapter 5 discusses a prototype chip for the MIMO accelerator. The chip test setup and measurement results are discusses in details. This chapter also compares the test results with state of the art ASIC implementations.

Finally this dissertation concludes with a list of contributions and possible future work.

2. MIMO Accelerator Hardware: Overview

The MIMO accelerator is a programmable device that targets MIMO decoding operations for MIMO-OFDM systems. To achieve this task, the MIMO accelerator is required to be able to implement the MIMO decoder algorithms that are reviewed in the previous chapter. This chapter will go through the hardware implementation of the MIMO accelerator that allows it to cover the mentioned algorithms.

2.1. Top View of the Accelerator

The accelerator follows the hardware architecture of a processor, as shown in Fig. 2.1. It is divided into a data path and a control path. The data path of the accelerator depends on a complex-matrix-based processing core. The processing core performs all the matrix operations that are necessary for MIMO decoding implementation and its pre-processing requirements. The processing core consists of four separate units: an addition/subtraction unit, a reciprocal unit, a multiplication unit, and a rotation unit. The next section will give more details about those four units.

The data path is completed by the data memory. This memory carries all the variables that are used as operands for the matrix operations and that are also used to store the operation results. Given the problem that the accelerator is designed to solve, all the variables in the data memory are matrices that are composed of $N_{rx} \times N_{rx}$ elements (where N_{rx} is the number of receive antennas), and each element is a complex-valued number. This data memory is connected to the processing core by two groups of multiplexers: the core-input switch and the memory-input

switch. The core-input switch delivers the right variables to the right inputs of the processing core based on the running instruction. And the same applies for the memory-input switch as it delivers the operation result from the right processing core output to the right ports of the data memory.

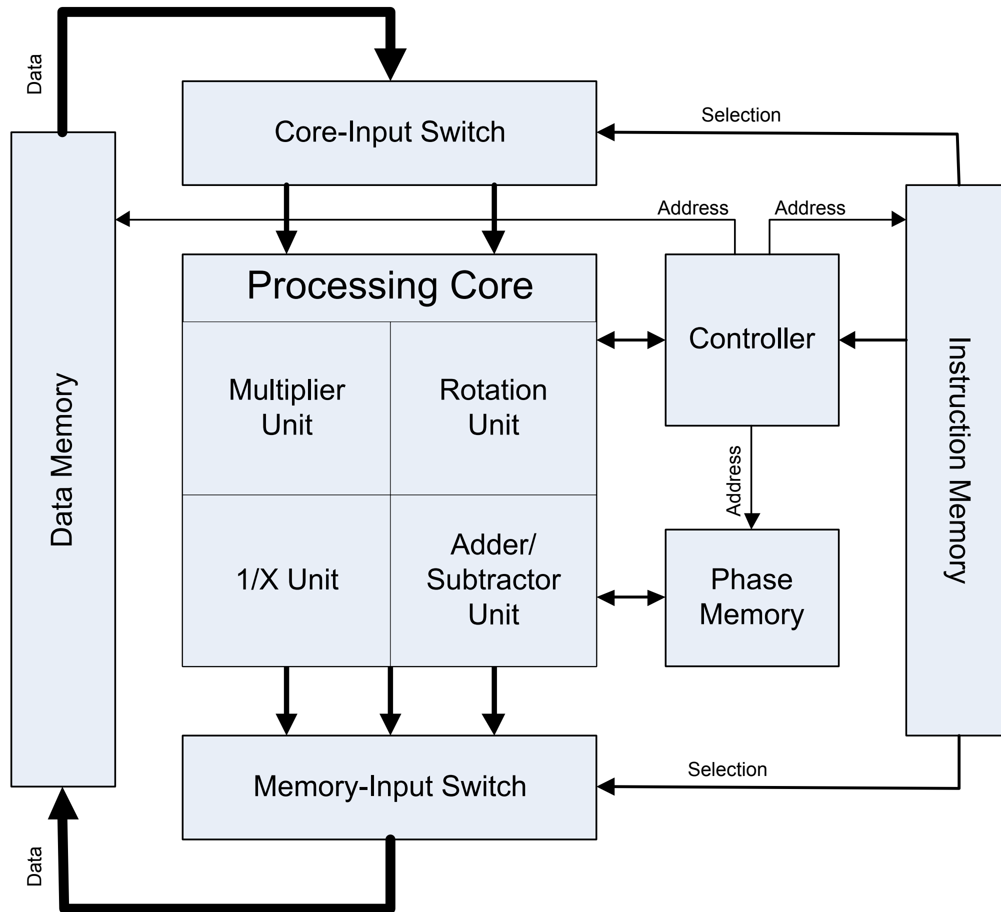

Fig. 2.1. MIMO Accelerator block diagram

The control path of the MIMO accelerator consists of two blocks: the instruction memory and the controller. The instruction memory carries the programs that are executed on the accelerator. Those programs are in the form of pre-decoded instructions that directly controls all parts of the data path. The choice to use pre-decoded instructions gives the accelerator user complete

flexibility in utilizing the accelerator resources. While a single instruction is wide (326 bits in the accelerator prototype), the number of instructions that are needed to run a complete MIMO decoder is limited (doesn't go higher than 64 instructions). The small number of instructions is a result of the parallel processing that is introduced in the processing units, as will be discussed later.

The controller, the second part of the control path, is mainly a Finite-State Machine (FSM) that controls the timing of memory read, memory write, and instruction execution. It controls the instruction counter, which is the address of the instruction memory. And it also controls the subcarrier counter, which is the address of the data memory (the relation between the OFDM subcarriers and data memory will be discussed in the next chapter). This FSM decides when to increment each counter based on the execution progress and the instruction being executed.

Fig. 2.1 shows three memories, both the data and the instruction memories have been discussed. The third memory is the *phase memory*. The phase memory is an SRAM that is limited to the storage of the rotation-unit phase outputs. This stored phase can be reused as the input to the internal blocks of the rotation unit for future instructions.

2.2. Processing Core

The processing core is the hardware block that is used to perform all the mathematical operations needed for the MIMO decoding algorithms that are reviewed in the previous chapter. It consists of four processing units. This section gives a detailed look into each of these units for four receive streams ($N_{rx} = 4$).

2.2.1. Addition and Subtraction Unit

First processing unit is the addition/subtraction unit (or simply the addition unit). A block diagram of this unit is shown in Fig. 2.2. The inputs of the addition unit are two sets of two vectors of N_{rx} elements each. This means that the addition unit performs two two-vector additions in one execution cycle, hence the eight adders shown in Fig. 2.2 for $N_{rx} = 4$. Each of the elements of these vectors is a complex number, leading to the fact that each addition block in Fig. 2.2 is in fact two adders. Based on a control bit from the pre-decoded instruction, the performed operation can be an addition or subtraction.

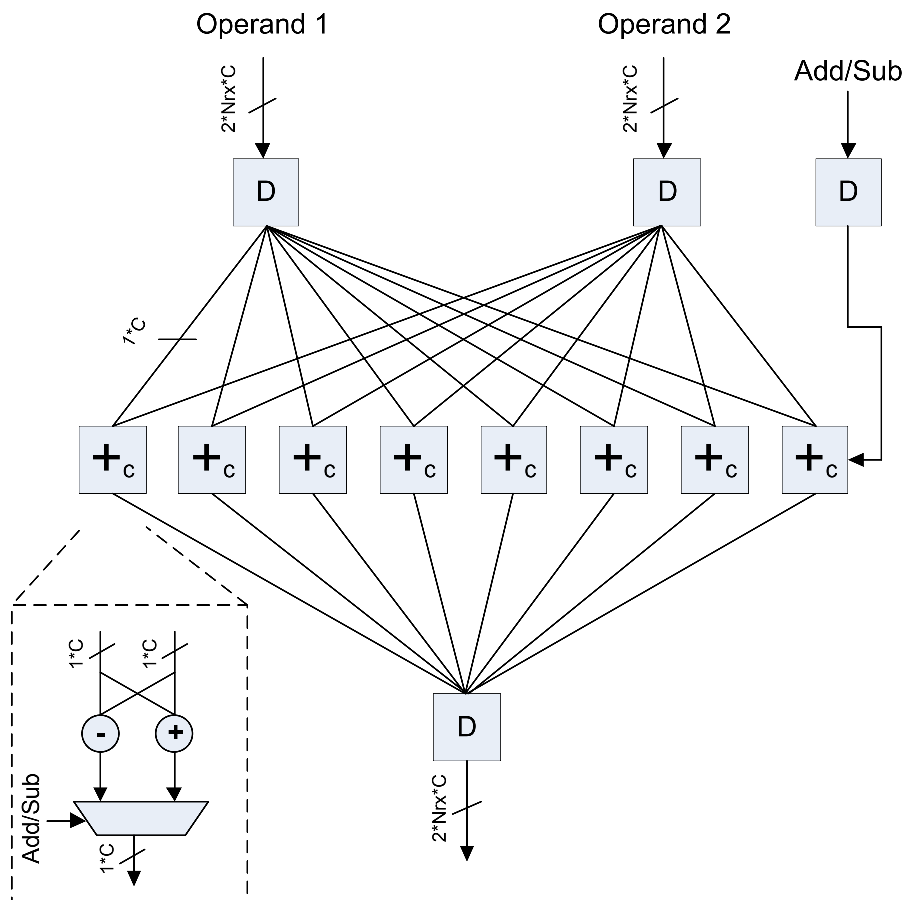

Fig. 2.2. Block diagram for the addition/subtraction processing unit for four receive streams

- N_{rx} is the number of receive streams (4 in this figure, but it might change as will be explained)
- C is the precision of one complex number

The addition unit is essential for matrix addition and subtraction, which is needed – for example – in the formulation of the MMSE matrix that should be inverted (equation 1.6). Another example is the metric calculation of ML and SD MIMO decoders.

2.2.2. Reciprocal Unit

The reciprocal unit is used to invert real numbers. Its main usage is in signal scaling, such as the multiplication by the inverse of the noise standard deviation in MMSE (equation 1.11). For four receive streams, it consists of four real dividers as shown in Fig. 2.3. Each divider directly implements a signed long division operation with the dividend fixed on +1. A reciprocal instruction controls the core-input switch to choose which variable matrix elements are used as operands. The divisors are routed from the data memory as the real parts or the imaginary parts of the routed complex numbers. The choice between real or imaginary parts to be the divisors is controlled by the pre-decoded instruction as well.

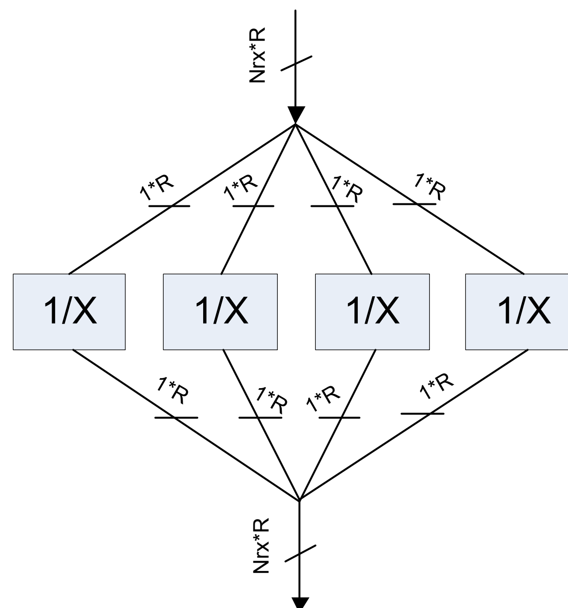

Fig. 2.3. Block diagram for the reciprocal processing unit for four receive streams

- N_{rx} is the number of receive streams (4 in this figure)
- R is the precision of one real number

2.2.3. Multiplication Unit

Multiplication operations are essential in any MIMO decoding algorithm. It is used for matrix formulation of linear decoders, as explained in equations (1.5) and (1.6). And it is also used to perform the actual decoding in the form of multiplying the received symbols and the W matrix in a linear decoder. In other decoders, such as ML, it is also essential. The required multiplication is always based on matrix operations. It can be vector-matrix multiplication or matrix-matrix multiplication. The structure of the multiplication unit performs this intensive computation by implementing dot products, as shown in Fig. 2.4.

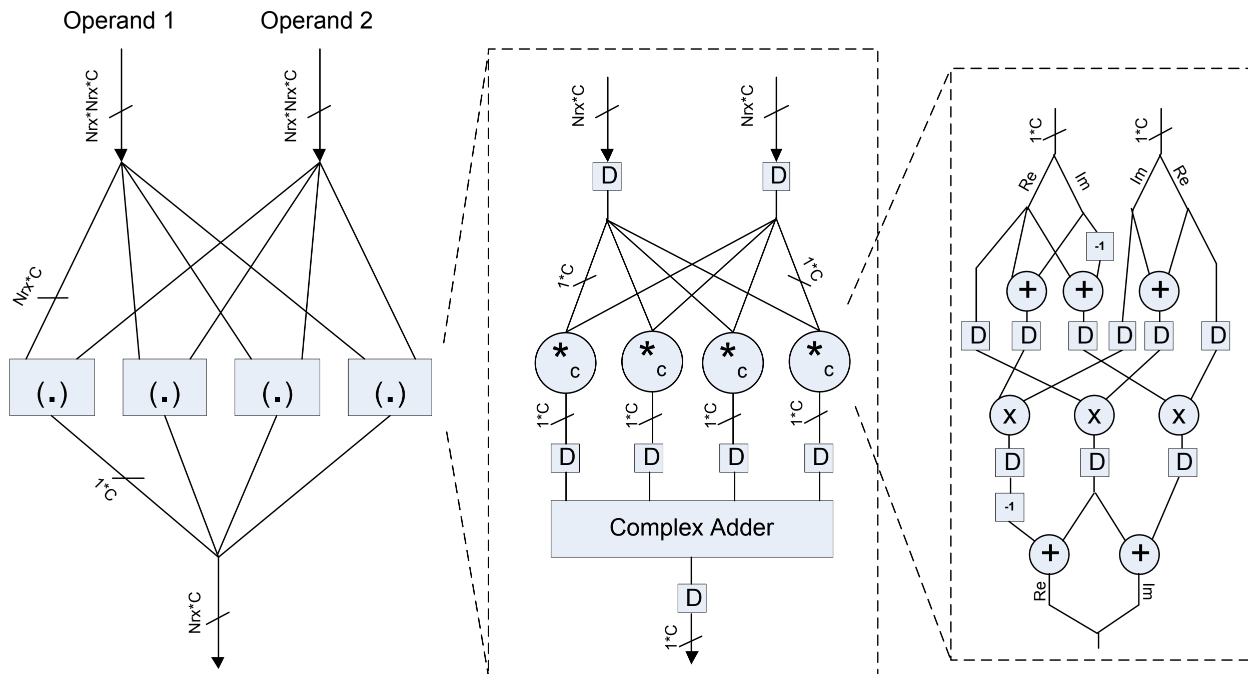

Fig. 2.4. Block diagram for the multiplication processing unit for four receive streams

- N_{rx} is the number of receive streams (4 in this figure)
- C is the precision of one complex number
- Re and Im stand for Real and Imaginary parts of a complex number

The multiplication unit is composed of N_{rx} dot product blocks. Each block computes the dot product of two complex vectors ($c = \boldsymbol{a}^T \boldsymbol{b}$, where $\boldsymbol{a}$ and $\boldsymbol{b}$ are column vectors and c is a complex number). And each vector consists of N_{rx} elements. The block diagram of Fig. 2.4 is based on $N_{rx} = 4$. Only three real multipliers are used to implement the complex multiplication instead of four, as shown in Fig. 2.4. This is explained in equations (2.1) to (2.3) as follows:

$$If\ x = a + i.b = (c + i.d)\times(e + i.f) \qquad (2.1)$$

$$a = c.e - d.f \quad and \quad b = d.e + c.f \qquad (2.2)$$

Then a and b can be rewritten as:

$$a = c.(e + f) - f.(c + d) \quad and \quad b = c.(e + f) + e.(d - c) \qquad (2.3)$$

This multiplication unit with this parallel processing allows a complete vector-matrix multiplication to be executed in one instruction. A complete matrix-matrix multiplication can be performed in N_{rx} instructions.

2.2.4. Rotation Unit

The rotation unit is responsible of all matrix factorization operations such as QR decomposition and SVD. These decompositions are used in MIMO decoding as a part of the decoding operation itself or as pre-processing operations to simplify the decoding task.

The hardware of the rotation unit is based on COordinate Rotation DIgital Computer (CORDIC) blocks, or more precisely complex CORDIC blocks as shown in Fig. 2.5. For four receive streams ($N_{rx} = 4$), four complex rotation CORDICs are used for vector rotation and one complex vectoring CORDIC is used for phase detection.

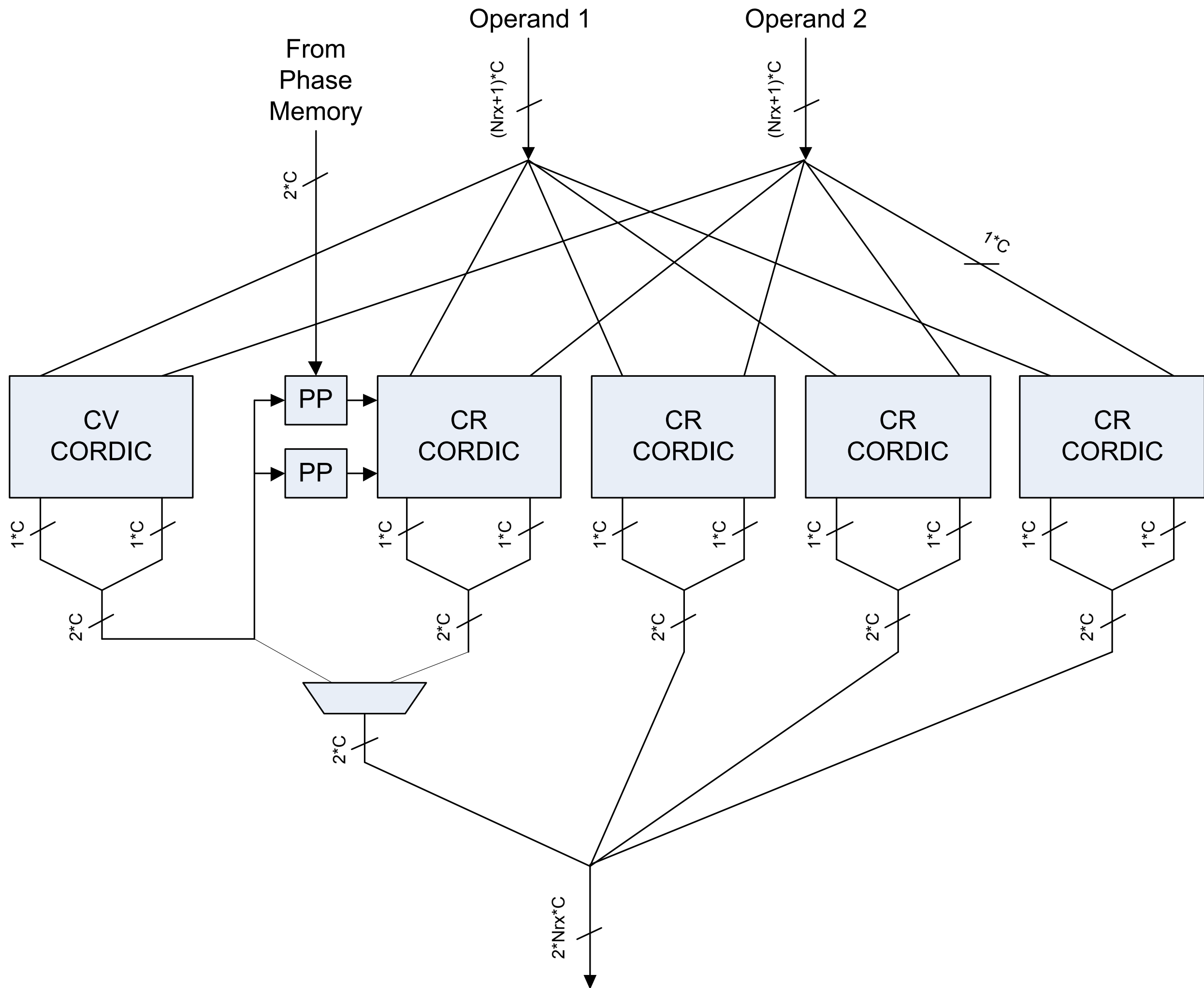

Fig. 2.5. Block diagram of the rotation processing-unit for four receive streams.

- CV stands for Complex Vectoring
- CR stands for Complex Rotation
- N_{rx} is the number of receive streams
- C is the precision of one complex number

A complex vectoring CORDIC consists of two vectoring CORDICs, as shown in Fig. 2.6. Those two vectoring CORDICs detect two phases. The first phase is an actual measured phase of complex number, and the other phase is computed from the absolute values of two numbers. On the other hand, a complex rotation CORDIC consists of three simple rotation CORDICs that use

the two phases of the complex vectoring to rotate a complex vector. A block diagram of complex rotation CORDIC is shown in Fig. 2.7.

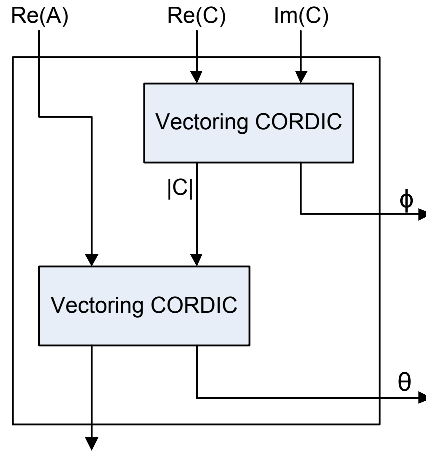

Fig. 2.6. Complex vectoring CORDIC

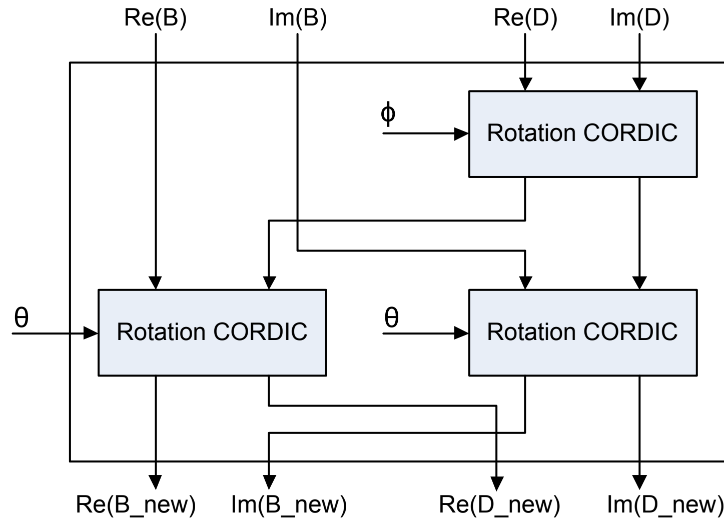

Fig. 2.7. Complex rotation CORDIC

The direct usage of the combination of a complex vectoring and a complex rotation CORDICs is in nulling an element of a complex matrix, which is the base of matrix decomposition using Givens rotations [26]. To null a complex element C of a 2x2 matrix – as shown in equation (2.4), the unitary transformation that is shown in the same equation is used.

$$\begin{bmatrix} \cos\theta & \sin\theta e^{-j\phi} \\ -\sin\theta & \cos\theta\, e^{-j\phi} \end{bmatrix} \begin{bmatrix} A & B \\ C & D \end{bmatrix}, where\ \theta = tan^{-1}(|C|/A)\ and\ \emptyset = \emptyset_C \qquad (2.4)$$

The complex vectoring computes $\emptyset$ and θ assuming A is a pure real number. For a 2x2 matrix, the transformation is applied on the two column vectors of the matrix; this results in the need of two complex rotation CORDICs. For a 4x4 matrix as in our rotation unit, the four elements of the unitary transformation of equation (2.4) replace the corresponding elements of a 4x4 unit matrix. The unitary transformation is then applied to the four column vectors of the 4x4 matrix, hence the need of four complex rotation CORDICs.

Fig. 2.5 also shows a hardware block that is called *phase processing* (PP). This PP is shown in Fig. 2.8. Part of the pre-decoded instruction controls this PP to select the source of the two phases that are used in the complex rotation CORDICs. This phase can be directly routed from the complex vectoring CORDIC. It can also be a one-instruction-delayed version of the complex vectoring output, which is useful in formulating the unitary matrices in the process of running a factorization algorithm. Phases read from the phase memory can also be used. The phases stored in the phase memory can be a result of a regular processing operation and it can be the output of an earlier instruction's complex vectoring operation. The PP gets the name from the fact that it performs the processing that is needed on the phase to decide the sign of each micro-rotation inside the three rotation CORDICs of each complex rotation CORDIC.

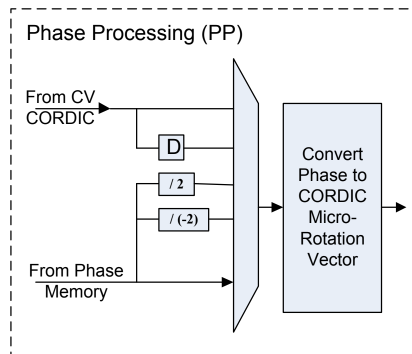

Fig. 2.8. Phase Processing (PP) block diagram as a part of the rotation unit.

The rotation unit is equipped with a group of multiplexers for signal re-routing. These multiplexers are controlled by the pre-decoded instruction and can be used to reconfigure the complex CORDICs to perform different operations. For example, the complex vectoring CORDIC signals can be re-routed to be as shown in Fig. 2.9. These modified connections allow the two simple vectoring CORDIC of the complex CORDIC to work independently to pick up phases of the complex inputs. This, combined by a re-routed version of the complex rotation, can be used for a unitary transformation for phase elimination of a complex matrix element. Re-routed versions of the complex rotation CORDIC are shown in Fig. 2.10.

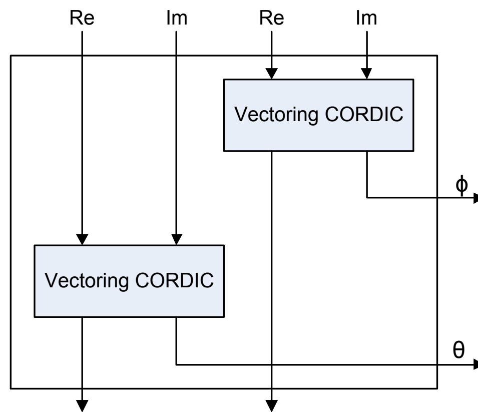

Fig. 2.9. A rerouted version of the complex vectoring CORDIC

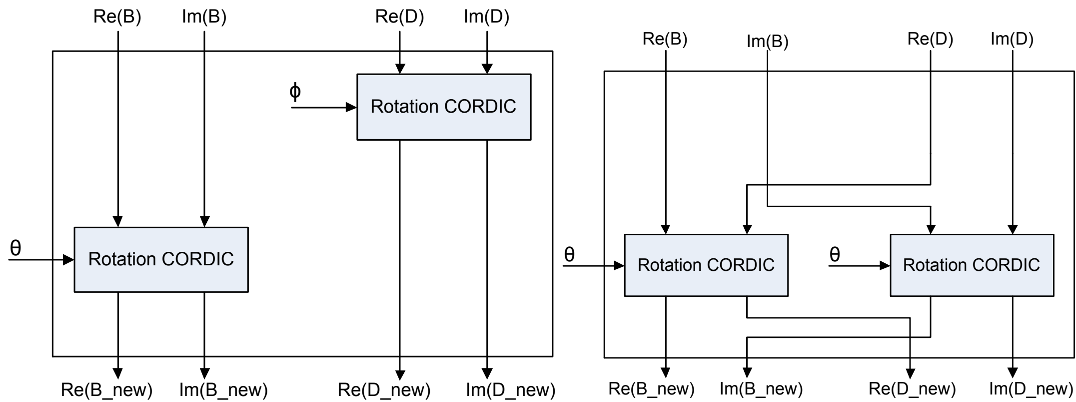

Fig. 2.10. Rerouted versions of the complex rotation CORDIC.

The left figure performs two completely independent rotations. The right figure can be used to rotate real vectors.

2.3. Dynamic Scaling

The output precision of both the multiplication and reciprocal units is double the precision of their inputs. This is to preserve the accuracy of the result. But for a programmable device like the MIMO accelerator, the precision has to be preserved for the limits of the memory size and the re-use of the same hardware for consecutive operations.

Instead of truncating a fixed part of the multiplication or division outputs, the accelerator uses dynamic scaling [16]. Using a group of OR gates, the dynamic scaling hardware detects the most significant non-zero bit among the absolute values of all elements of the vector output of the current operation (whether this operation is a multiplication or division). The higher order bits are then truncated in all elements, which guarantees that at least one vector element reaches the most significant remaining bits with a non-zero bit. To keep the output with the same precision of the input, the remaining bits to be truncated are taken from the least significant side. This operation guarantees no more than 3dB quantization loss using 16-bits of precision for matrix-matrix multiplication, compared to floating point operation. Instructions control whether the dynamic scaling is used based on a single instruction result, or based on the result of consecutive instructions.

2.4. Memory Switches

The memory switches are responsible for delivering the operands from the data memory to the processing core, and delivering the results from the processing core to the data memory. Fig. 2.11 shows block diagrams of the two switches.

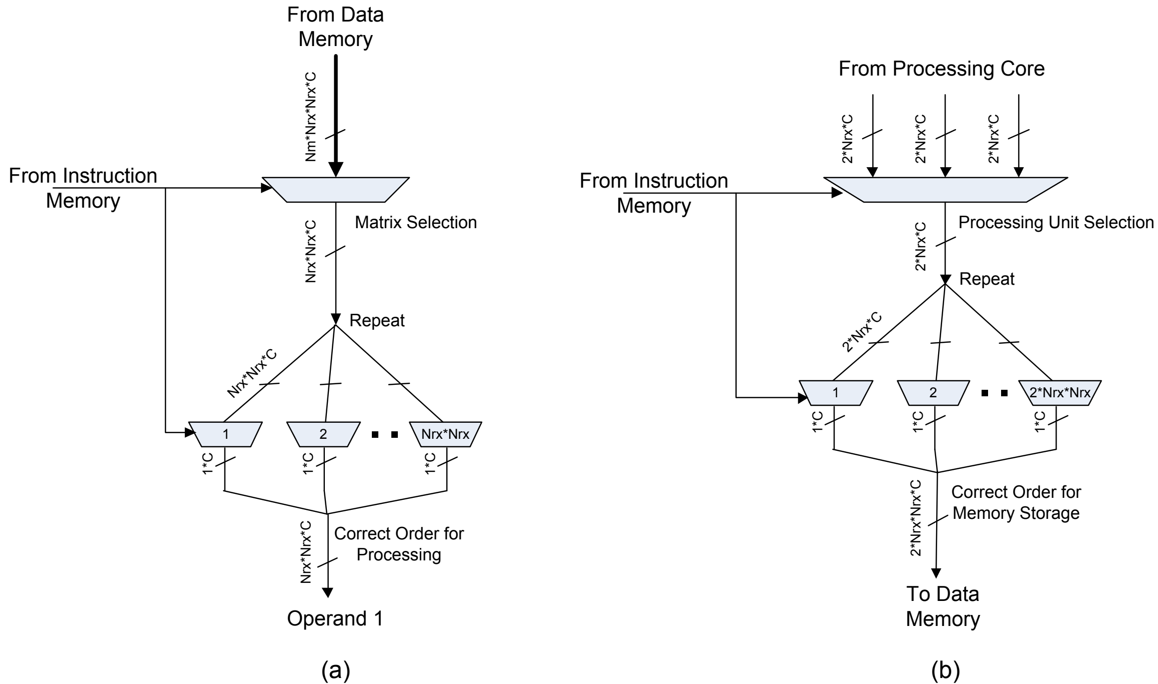

Fig. 2.11. Memory switches: (a) Core-Input Switch, Right, (b) Memory-Input Switch

- Nm is the number of variable matrices in memory
- Nrx is the number of receive antennas
- C is the precision of one complex number

The core-input switch, as shown in Fig. 2.11a, is a two-level multiplexing circuit. The first level takes the data memory output as an input. This data memory output is a group of all complex matrix variables. The first multiplexing level selects one variable matrix and delivers its $N_{rx}xN_{rx}$ elements to the second multiplexing level. The second multiplexing level has a number of multiplexers that is equal to the number of complex numbers in one processing core operand. That means it has $N_{rx}xN_{rx}$ multiplexers, each of those has its output connected to one complex number of the operand. The inputs of each multiplexer of the second level are all the elements of the matrix that is chosen by the first level. This allows delivering any matrix element to any operand output. This permits operations on row vectors, column vectors, matrix diagonals, or any arbitrary combination of complex numbers. The selection lines of the two multiplexing levels are part of the pre-decoded instruction, which gives the programmer a complete flexibility

on choosing operands. The complete core-input switch of Fig. 2.11a is repeated twice in hardware, one switch for each operand.

The memory-input switch has a similar structure, as shown in Fig. 2.11b. Its first level of multiplexing chooses one of three processing core outputs: the addition/subtraction unit output, the rotation unit output, or the dynamic scaling output (which is the output of the multiplication unit or the reciprocal unit). The second level of multiplexing contains a multiplexer for each complex number of two matrices of the data memory input. The choice of those two matrices is controlled by the instruction. Each multiplexer selects any complex number from the chosen unit output. This again gives the flexibility of arranging the result as a row vector, column vector, matrix diagonal, or an arbitrary arrangement.

3. MIMO Accelerator Hardware: Implementation Challenges and Tradeoffs

Chapter 2 gave an overview of the accelerator hardware implementation and how it can be used to implement MIMO decoding. It also went through some of the flexibility that is given to the programmer through the pre-decoded instructions. This chapter dives deeper into the hardware implementation by showing some of the design challenges and how they are solved.

The main performance metric of the accelerator is energy consumption. This was always kept in mind when the hardware is implemented. Solutions for the hardware challenges were based on minimizing the energy consumption whether it was by reducing the power consumption keeping the throughput, or by reducing the area keeping the same flexibility of the accelerator, which in turn allows smaller programs and hence lower energy consumption.

This chapter will discuss six different implementation challenges and the tradeoffs that are inspected for their hardware solutions.

3.1. Instruction Fetch and Pipeline

The MIMO accelerator is a pipelined processor, which means that the accelerator accepts a new instruction every clock cycle. Fig. 3.1 gives an illustration on how the accelerator handles the instruction execution. Every clock cycle an instruction enters the accelerator pipeline starting with an instruction fetch, which is the instruction read from the instruction memory. The second stage in instruction processing is the operand read stage, which is reading the data memory

contents that are required by the instruction and routing it to the operands through the core-input switch. The third stage is the execution, where the operands go through the processing core. This execution stage takes 68 clock cycles to finish as the processing core is deeply pipelined for higher throughput. The fourth and final stage for instruction processing is the result write-back. In this stage the processing core result is routed through the memory-input switch to the data memory to store the result. During the first clock of an operand read for an instruction with index n, the next instruction with index n+1 enters the fetch stage. This behavior of the accelerator (and pipelined processors in general) means that, after the initial latency, all the four stages of instruction processing will be running simultaneously on different instructions. A side note here is that there isn't a decode stage because the accelerator uses pre-decoded instruction. This way the user has full access to the accelerator hardware.

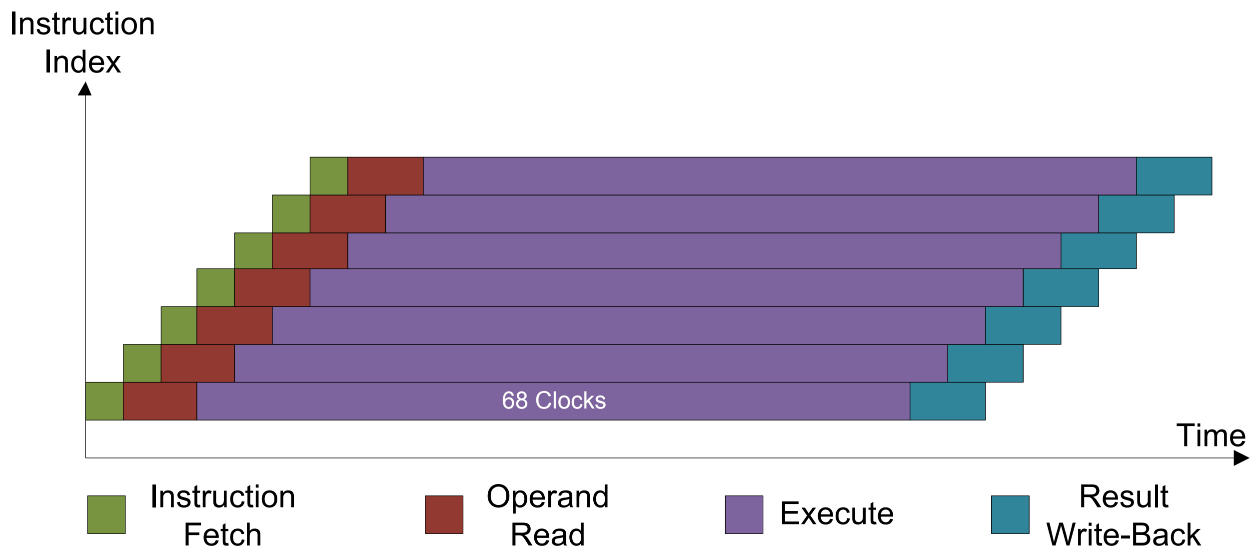

Fig. 3.1. Accelerator instruction pipeline with four stages

The challenge in the hardware implementation of this processing cycle arises from the bit width of the instruction. The pre-decoded instructions of the accelerator are 326 bits wide, as shown in

the instruction structure in Fig. 3.2. As a new instruction enters the accelerator pipeline, the old instruction should be kept registered throughout its processing stages. This is called pipelining the instruction by delivering it from one register to the next, and the number of registers is the number of the pipelining clock cycles. As the pipeline cycles are long (the execution stage is 68 clock cycles) and the instruction is wide, those instruction pipeline registers occupy a huge area.

Flow Control 16b	Proc. Config. 15b	Operand Selection 132b	Result Destination 99b	Write Mask 64b

Fig. 3.2. Accelerator pre-decoded instruction structure

The solution of this waste in area lies in the structure of the instruction that is shown in Fig. 3.2. The *processing configuration* part of the instruction is the only part that is needed during the execution stage; it controls the internal operations of the execution like the re-routing multiplexers. This means that the processing configuration part of the instruction has to be pipelined throughout the execution stage, which is acceptable given the small width (15 bits).

The widest part, the *operand selection*, is only needed in the operand read stage. Its 132 bits are divided as follows: 2 bits per operand to pick up a variable matrix, and 16x4 bits per operand to select which element of the memory matrix (16 elements per matrix) is routed to which element of the operand (16 elements per operand). This operand selection part can be dropped from the long pipeline.

The problem remains in the *result destination* and the *write mask* parts. These two parts are combined for 163 bits and they are needed in the result write-back stage. The result destination

part of the instruction (99 bits) is used to route the processing core output to the data memory input through the multiplexers of the memory-input switch. These 99 bits are divided as follows: 2 bits to choose which processing unit of the three results is used, 2x16x3 bits are used to select which result element (out of 8 elements) is routed to which variable matrix elements (covering two variable matrices as mentioned earlier), and 1 bit to pick-up which two matrices out of the four are the target. The write mask (64 bits) is used to control which elements of the data memory are overwritten by the result, and it will be discussed in the data memory section of this chapter. Pipelining those 163 bits throughout the long processing cycle is still a waste of area.

Based on the fact that the waste in area is only caused by the write-back stage, we introduced an extra processing stage to avoid this problem. Fig. 3.3 shows the new instruction pipeline with the five processing stages. The extra stage is a new instruction-fetch stage that is performed directly before the result write-back stage. Instead of pipelining the instruction from the original fetch, we fetch it again naming the original fetch the *read fetch* and the second fetch is named the *write fetch*. By introducing this write fetch stage, the instruction memory has two be a dual-port memory to allow the two fetches to run simultaneously. But, on the other hand, the long and wide pipeline registers are avoided.

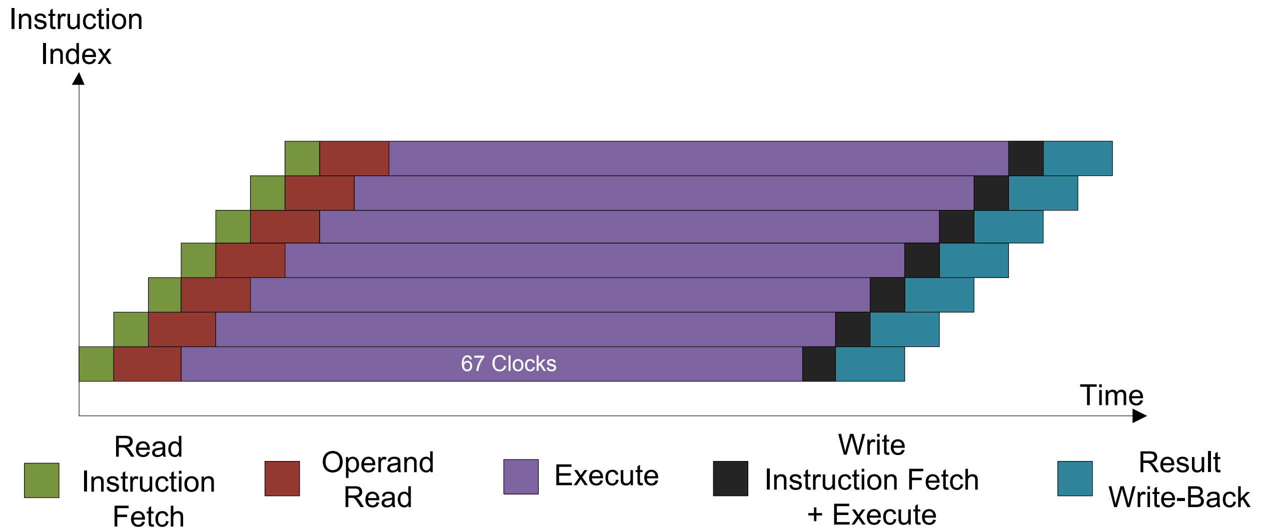

Fig. 3.3. Accelerator instruction pipeline with five stages after introducing the extra fetch stage

This solution introduced an area saving of 5.3% of the complete accelerator area (this includes all memories ad processing units). This saving in area didn't introduce any more latency to the instruction processing. The write-fetch stage doesn't depend on the execution stage result. The write fetch is performed simultaneously with the last clock cycle of the execute stage.

3.2. Controller Counters Pipeline

As mentioned in the previous chapter, the controller is a FSM that controls the instruction and subcarrier counter. These two counters are the addresses for the instruction and data memories respectively (the relation between the data memory and the OFDM subcarriers is discussed in the next section). The subcarrier counter has to be present for a certain instruction execution during the operand read and the result write-back stages. That means it has to be pipelined throughout the long processing cycle. The same goes for the instruction counter; by introducing the write

fetch stage, the instruction counter has to be pipelined during the processing cycle to reach the write fetch stage.

The complete controller area, including the FSM hardware and the counters, is only 3.5% of the area of the added registers for this pipeline. This obviously is a waste of hardware area. Given this small area of the controller, we introduced the solution of this waste. The solution is to repeat the complete controller hardware, including the FSM. That is to say that the accelerator now has two instances of the controller, one to control the timing and the addresses of the read fetch and the operand read. This controller is called the *read controller*. The other controller is to control the timing and addresses of the write fetch and the result write-back. This second controller is called the *write controller*. The two controllers are shown in Fig. 3.4.

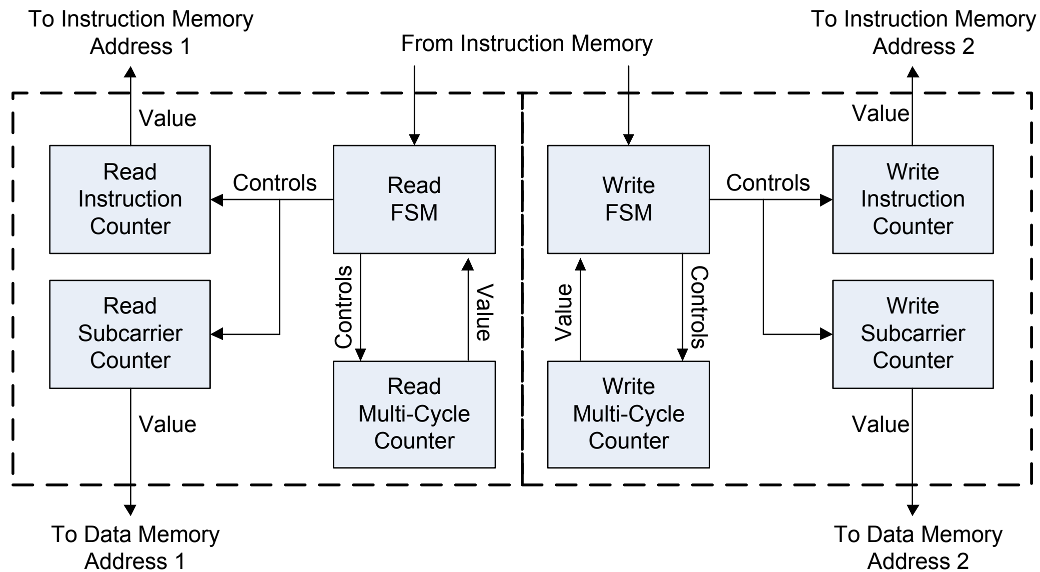

Fig. 3.4. The two instances of the controller: read controller (left) and write controller (right)

The state machines of the two controllers have the same hardware structure and state tables, but the write controller FSM is delayed from the read controller state machine. The write controller

is kept in a reset state for the duration of the latency of the processing cycle of the first instruction until this first instruction reaches its write fetch stage.

3.3. Data Memory Structure

The accelerator operation is based on complex matrix and vector operands and results. The usage of a regular linear memory structure for complex numbers, as shown in Fig. 3.5a, is not efficient. A linear memory will result in a huge latency to form the inputs that are required for a multiplication operation as an example. This parallel load requirement is added to the flexibility of forming row, column, diagonal, and arbitrary vectors to lead to the idea of using a register for the complete data variables.

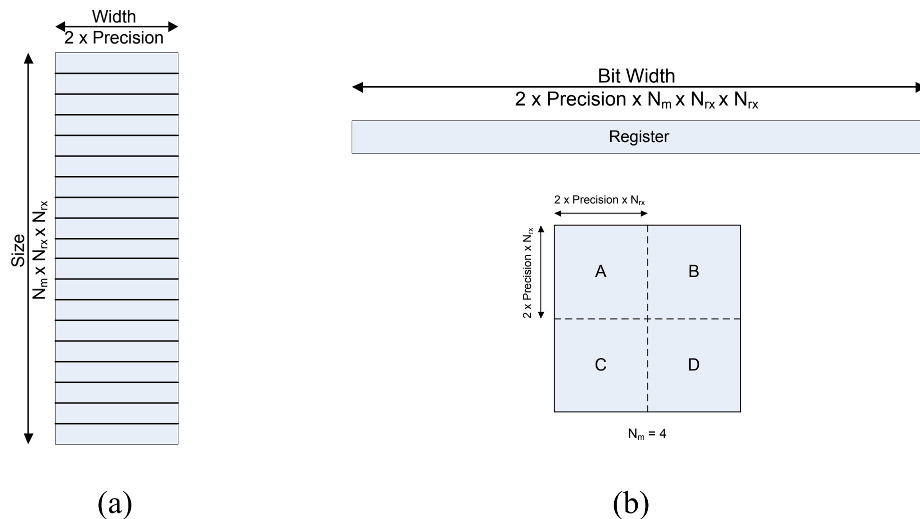

Fig. 3.5. (a) Linear data memory structure, (b) One register for the data memory

- N_{rx} is the number of receive antennas
- Precision is the bit width for a real number
- N_m is the number of variable matrices

Fig. 3.5b shows the design of the complete data set as one register. The complete data set is N_m variable matrices, each matrix is $N_{rx}xN_{rx}$ where N_{rx} is the number of receive antennas, and each

matrix element is a complex number of bit width 2xPrecision (Precision is the bit width of a real number throughout the MIMO accelerator hardware). This structure allows complete access to all the data variables with the help of the two switches (core-input and memory-input switches). This big register is logically divided into matrices called A, B, C, and so on. Fig. 3.5b shows this logical division for N_m=4.

The complete data set, as shown in Fig. 3.5, is repeated for every OFDM subcarrier. The subcarriers are independent, this allows the arrangement of the variables of all subcarriers in one data memory, as shown in Fig. 3.6. Every memory location is dedicated to a subcarrier. This is a return to the linear memory but with every memory location carrying a very wide word for the complete data set of a subcarrier. A memory location carries 2048 bits for four variable matrices (N_m=4), four receive antennas (N_{rx}=4), and a 32 bits complex number (Precision=16).

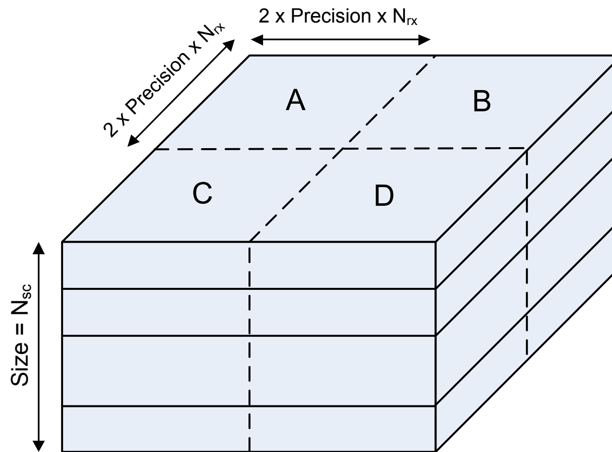

Fig. 3.6. Proposed data memory structure (for N_m=4)

- N_{sc} is the number of OFDM subcarriers
- N_{rx} is the number of receive antennas
- Precision is the bit width for a real number
- N_m is the number of variable matrices

This memory arrangement is suitable for the required data access parallelism and flexibility, but it causes a problem in the result write-back stage. A result of an instruction is a combination of complex vectors that should replace only a small part of the data variables, based on the result destination part of the pre-decoded instruction (Fig. 3.2). Writing a result for a subcarrier in the memory structure of Fig. 3.6 will replace the complete data set of this particular subcarrier leading to a loss of essential data for the remaining instructions.

Three solutions are possible for the result write-back problem. First solution is to pipeline the complete data set of a subcarrier from the operand read stage to the result write back stage. The result then replaces the specific required locations in the last pipeline register. This last pipeline register is then written to memory keeping the un-overwritten variables safe. This completely solves the problem but causes an overhead of registers that are too wide (2048 bits) and too many (the pipeline length is the more than 68 cycles). The area penalty is 50% of the complete accelerator area.

The second possible solution is to avoid this pipeline with the same design technique used in the instruction pipeline problem. This is to re-read the data memory location again directly before the result write-back stage, replace the target elements, and then write the data back in memory. This solution requires a triple port memory. The operand read and result write-back stages are running simultaneously (pipelined processor structure); adding another data memory access operation requires a third port. Triple port memories are built by doubling the memory into two dual-port memories, write in both, and then read separately. This obviously is a waste of area; especially as the data memory is about 44% of the accelerator area (as will be detailed in Chapter

5). Another way of implementing a triple port memory, as used in quad-port FPGA block memories [27], is by overloading one port with double-frequency access and using this port as two ports by time multiplexing. This design concept has drawbacks in terms of power and reliability as will be discussed in section 3.5.

The third and implemented solution is to give a small waste in area by dividing the data memory block into sectors, as shown in Fig. 3.7. A sector is an independent memory block that carries one matrix element (a complex number) for all subcarriers. This is a combination of linear memories leading to a small waste in area due to the overhead of each memory (sense amplifier, decoders, etc.) but allows complete control over which data elements are replaced. All memory sectors share the same address (read and write addresses for the two ports), but their write-enables are separate. The write enables are part of the pre-decoded instruction (write mask in Fig. 3.2.

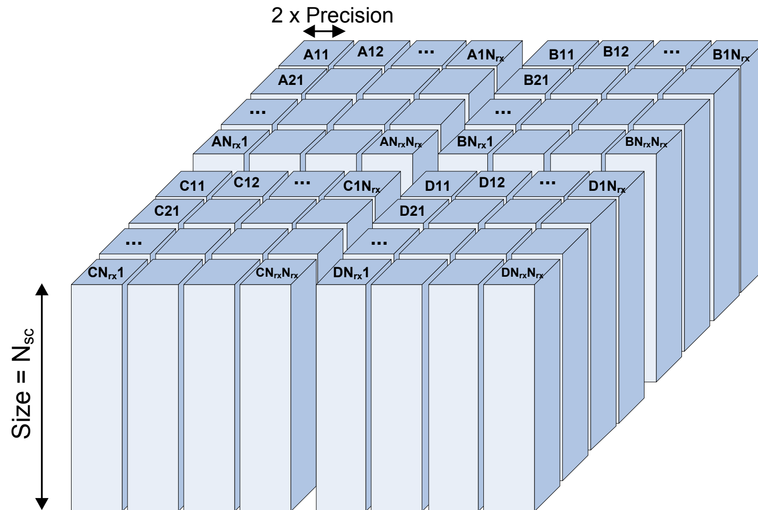

Fig. 3.7. Implemented sectored data memory (for N_m=4)

- N_{sc} is the number of OFDM subcarriers
- N_{rx} is the number of receive antennas
- Precision is the bit width for a real number
- N_m is the number of variable matrices

3.4. Processing Core Input-Gating

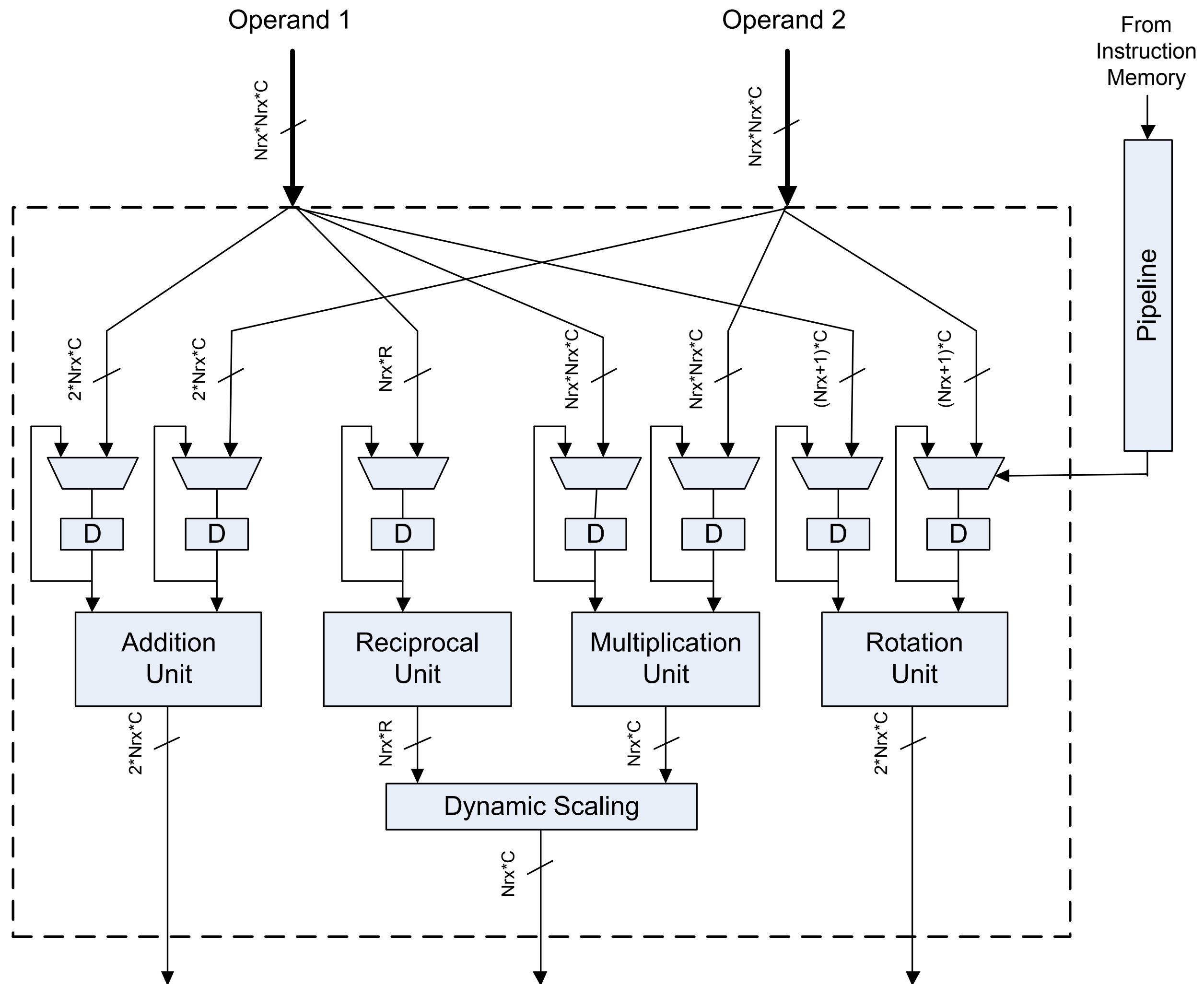

Fig. 3.8. Processing core after gating the four processing units

- N_{rx} is the number of receive antennas
- C is the bit width for one complex number

By performing complex mathematical operations with matrix operands in a deep-pipeline fashion, the four processing units are the most power hungry blocks in the accelerator. In the same way a regular arithmetic-logic unit (ALU) is designed, if the inputs to the four processing units are connected directly to the outputs of the core-input switch, the four units would be processing the two operands and just one result is selected using the memory input switch. This

is obviously a waste of power. To avoid this extra power consumption, extra hardware is introduced as shown in Fig. 3.8. Each processing-unit input is multiplexed between its old value and the new value coming from the core-input switch. Moreover, a small portion of the result destination part of the instruction is pipelined (from the read fetch stage) to provide the selection lines for the gating multiplexers.

The decision to add extra hardware in the operands path is non-trivial. The wide operands formed as complex matrices require wide registers and numerous parallel processing elements. This extra hardware increases the processing core area by 2.5% in addition of adding an extra latency stage. This cost of area and latency is acceptable as the resulting saving of power consumption is 15% of the power consumption of the processing core

3.5. System Integration

One of the design challenges that face the MIMO accelerator is how it will be integrated in a complete wireless receiver (a top level to the accelerator). As the data memory is the place where the initial values and the processing results are stored, such integration will be possible by granting the wireless receiver direct access to the data memory. This access will enable the receiver to exchange data with the MIMO decoder that is implemented on the accelerator.

Using a dual-port SRAM for the data memory is a necessity to achieve the required memory access. But, being a pipelined processor, the accelerator actually uses a dual-port SRAM to allow reading operands from a port and writing results through another port simultaneously. This means the accelerator needs a third data-memory port for top-level access. As mentioned in

section 3.3, triple port memories can be built by using two dual-port memories and perform the write operations on the two memories simultaneously, and perform two independent read operations. This way of implementing a triple port memory causes a huge waste of area given the big portion the data memory is compared to the accelerator (44%).

Another way of implementing a third port for the data memory to allow top-level access is to overload the data memory ports. Port overloading means using double the processing frequency for the memory ports, and time multiplex a single port between two tasks. The memory hardware is still a dual-port memory, which can be generated by a memory compiler, but the effective result is a higher number of ports (up to four effective port). Fig. 3.9 shows an illustration to such a design for the MIMO accelerator data memory.

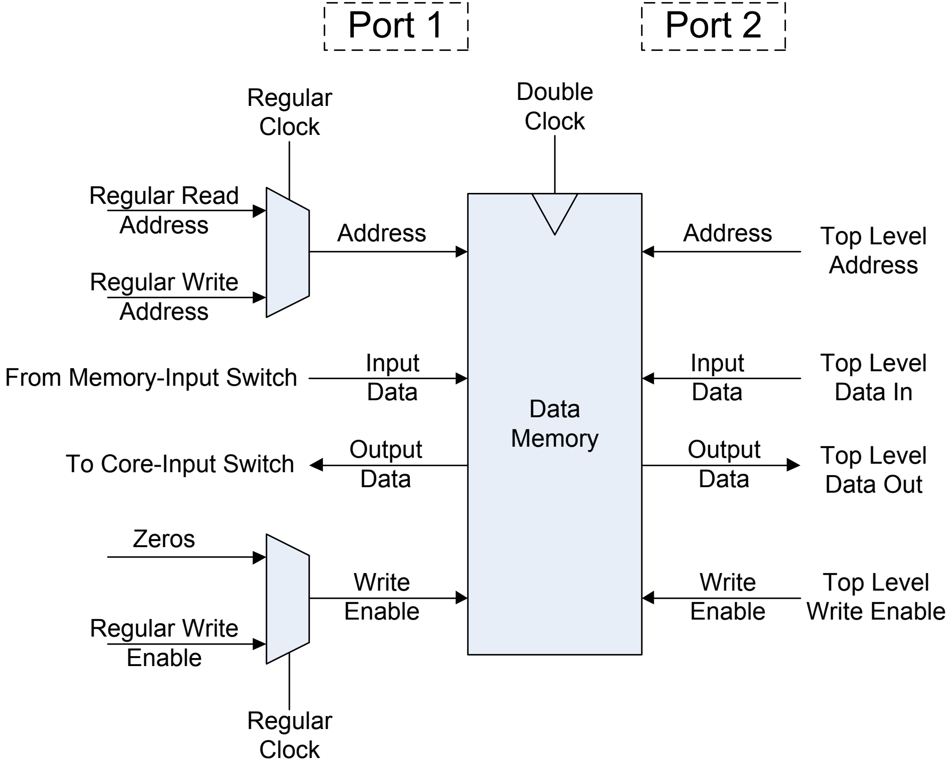

Fig. 3.9. Overloading one memory port to allow top-level access to the data memory contents

Port 1 in Fig. 3.9 runs on double the accelerator frequency. The address and write enables of this port are multiplexed between the operand-read and the result write-back processing stages. Port 2 is dedicated to the top-level access. This implementation gives an advantage of complete flexibility for the top-level to access the data memory. The top-level access doesn't interrupt the accelerator execution, and doesn't require any controls from the accelerator controller for memory access.

The port-overloading solution gives the required access for the top-level design but it has two major drawbacks. The first is the required synchronization between the regular accelerator clock and the memory port 1 clock. The frequency of the memory port 1 clock has to be precisely twice the processing clock. The phases of the two clocks also have to be perfectly aligned due to the fact that the two edges of the slow processing clock are used for memory access in the data memory. This synchronization requirement puts a limit on the maximum clock speed that the accelerator can work on in order to guarantee a high reliability for the final fabricated circuit.

The second drawback is the unnecessary increase in the power consumption that results from the usage of double the clock frequency in part of the system. This power consumption is unnecessary because this higher clock speed is not part of the accelerator processing requirements, but it is only to provide the top-level design an access to the data memory. As the accelerator goal is to achieve energy consumption close to dedicated designs, the extra power consumption that is not needed for processing has to be avoided.

To avoid the synchronization and extra power consumption issues, the flexibility of the top-level access has to be sacrificed. Fig. 3.10 shows the memory access circuit that is implemented in the MIMO accelerator.

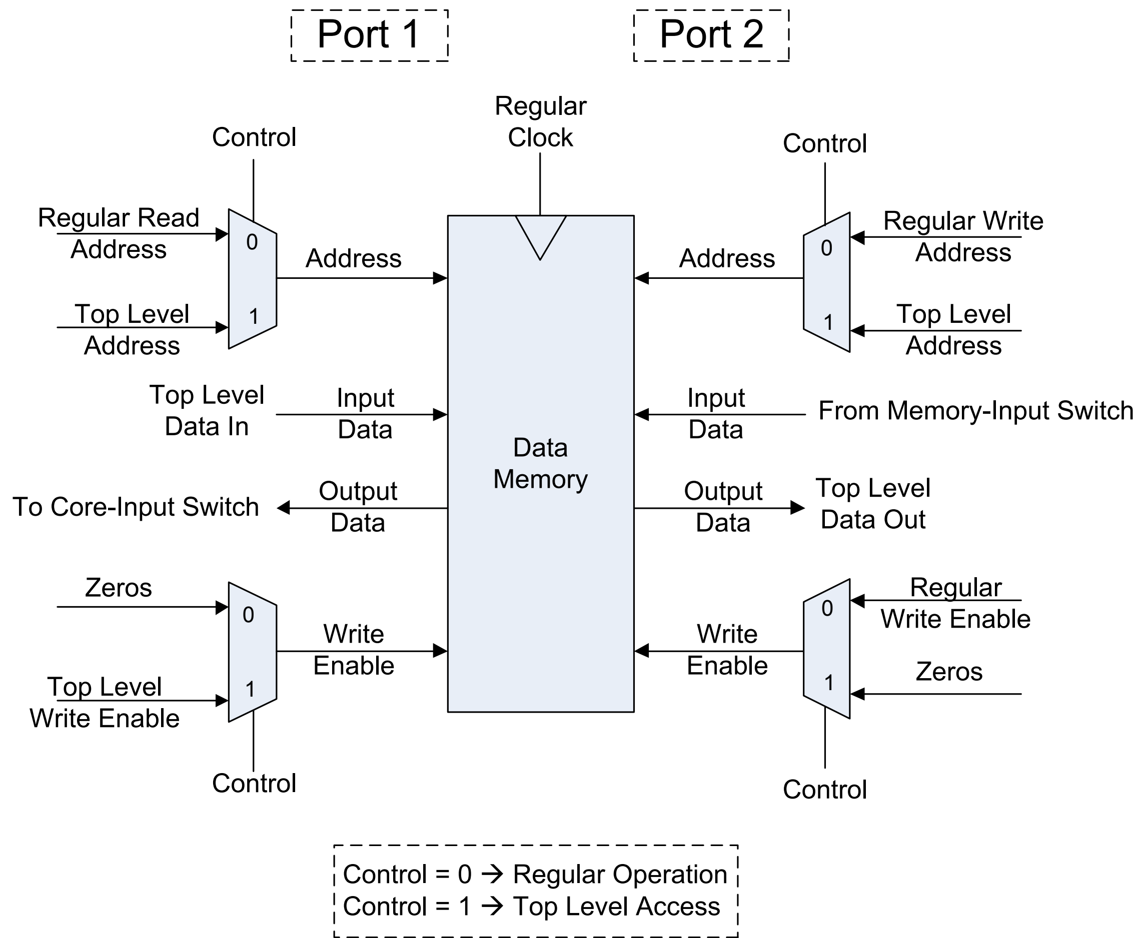

Fig. 3.10. Multiplexing the two data memory ports between regular operation and top-level access

As shown in Fig. 3.10, the two data memory ports are multiplexed between the regular accelerator operation (operand-read and result write-back) and the top-level access. For the top-level design to access the data memory to read previous results or to initialize the memory with new data, the accelerator operation has to be interrupted. This is a degradation of top-level access

flexibility but results in the use of one clock for the complete accelerator design, avoiding the extra power of a double-frequency clock and avoiding the reliability issues.

The multiplexing between the regular operation and top-level access is designed such that the data lines are not multiplexed. Regular operation writes through port 2 and reads from port 1, while the top-level access writes through port 1 and reads from port2. This saves the wide multiplexers that would have been used for the wide data lines on the two memory ports.

3.6. Multi-Algorithm Switching

The MIMO accelerator is designed to be able to run different algorithms and to give the system designed the ability to switch from an algorithm to another. Based on the way the accelerator is used, this algorithm switching might be needed on the fly, which means switching between algorithms without the need to re-write the instruction memory contents with a new program. Even for one algorithm, multiple programs might be needed. For example, in linear MIMO decoding (such as MMSE) one program is used for channel-matrix processing (inversion) and another program is used for the actual decoding (multiplying the received symbol by the result of channel-matrix processing). To switch between two programs that implement a single algorithm, a receiver wouldn't stop processing to rewrite the instruction memory.

To allow program switching, branching in the software (the program in the instruction memory) is not enough. The top-level hardware has to be the controller of such switching. To allow the top-level hardware to control program switching, the two most significant bits of the instruction memory address are given as inputs to the MIMO accelerator, as shown in Fig. 3.11. This means

that the instruction memory is logically divided into four sections and the top-level hardware controls which section is being executed. Given the parallel processing in the accelerator processing-core, a section can be as small as 64 instructions and be enough to carry the longest possible programs (more details about programming in Chapter 4).

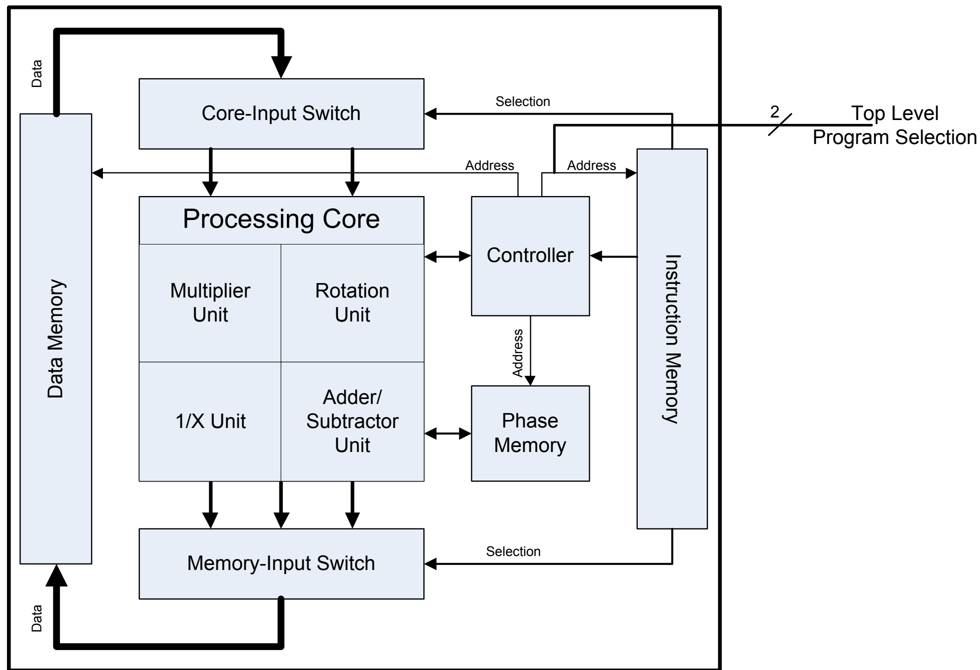

Fig. 3.11. Top-level control over the running program through the address of the instruction memory

4. MIMO Accelerator Design Flows and Programming

After introducing the MIMO accelerator hardware architecture and design concepts, this chapter goes through the software side of the MIMO accelerator. This chapter focuses on programming the MIMO accelerator by introducing its special programming language and tool flow.

Before going into the depths of the software-side of the accelerator, this chapter reviews one more aspect of the hardware: the HDL parameters. And introduces the design flows that the accelerator can be part of to have a complete MIMO decoder implemented on chip.

4.1. Hardware Parameters

The MIMO accelerator hardware description in VHDL is highly reconfigurable using a set of VHDL parameters. Those parameters control every aspect of the accelerator such as precision, number of OFDM subcarriers, number of MIMO receive streams, number of matrix variables, presence or removal of individual processing units, and etc. Table 4.1 shows a brief description of the main parameters.

Parameterizing the description of the accelerator gives the system designer the ability to map the accelerator hardware to a particular standard. If the system designer will use the accelerator for a 2x2 802.11n system, the parameters can be used to map the hardware for 2x2 802.11n. And if the design is for 8x8 LTE system, the parameters can be set for that. And also if the accelerator will

be used for multi-standards, the parameters can be mapped to the most complex one and used to run any standard.

Table 4.1. MIMO accelerator HDL parameters and their values for the ASIC prototype

Parameter	Description	Prototype Value
N_{rx}	Number of data streams (vector size)	4
N_m	Number of data-memory variable matrices	4
N_{sc}	Number of OFDM subcarriers (data memory depth)	52
P	Fixed point precision per rail	16
N_{ins}	Maximum possible number of instruction (instruction memory depth)	256
T_a, T_r, T_m, T_v	Availability of individual processing units	1, 1, 1, 1
M_v/M_d	0/1: All processing units have the same number of outputs 1/0: All processing units have the same number of inputs	0/1
D_r/D_q	0/1: A complete data matrix can be routed to an operand 1/0: One vector only can be routed to an operand	0/1

4.2. MIMO Accelerator Design Flows

Based on the HDL parameters of the previous section and the tools that will be introduced later in this chapter, this section introduces two design flows that employ the flexibility and configurability of the accelerator.

4.2.1. Hardware-First Design flow

Based on the programmability and the parallelism of the MIMO accelerator hardware, this design flow is considered the basic way of using the MIMO accelerator. A flow chart of the hardware-first design flow is shown in Fig. 4.1.

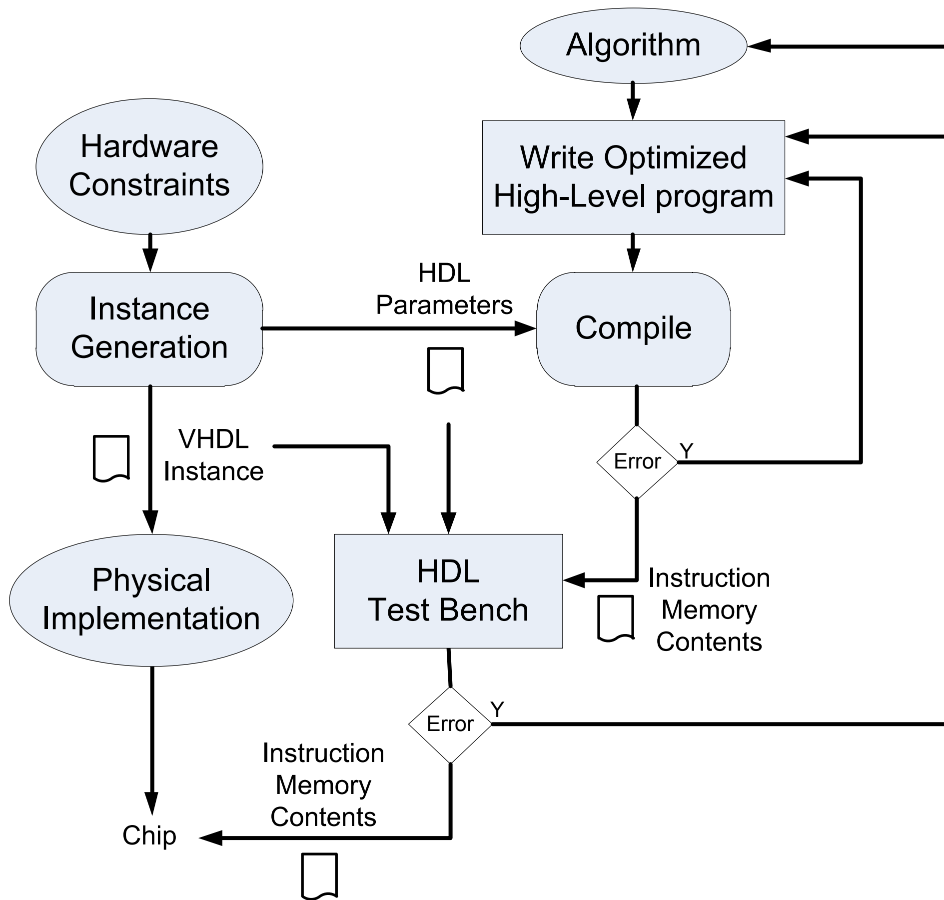

Fig. 4.1. Hardware-first design flow to utilize the MIMO accelerator programmability

As its name indicates, the hardware-first flow starts from the hardware constraints of the system – such as chip area, power budget, and clock period that are allowed for the accelerator. The HDL parameters are chosen based on these hardware constraints such that the accelerator processing is as powerful as the constraints permit. For example, if the area allows a 4-antenna and 1024-subcarrier accelerator, then the parameters are chosen to allow that configuration without any saving in the assigned area by choosing a smaller configuration.

The *instance-generation* tool (that will be introduced later in this chapter) uses those chosen parameters to generate a synthesizable VHDL description of the accelerator and its test bench. The generated design goes through the physical implementation to produce the final chip. By

starting by the hardware constraints, this accelerator chip is as powerful as possible in terms of throughput and standard variations support.

After the chip is implemented, algorithms that will be running on the accelerator are written in the high-level programming language of the accelerator, as shown in the right side of Fig. 4.1. Those programs are compiled through the accelerator *compiler* to generate the instruction memory contents. The programming language and the compiler will be discussed later in this chapter. After testing the programs on the accelerator generated test bench, they can be written on the on-chip instruction memory. The instruction memory in this case has to be implemented as a RAM.

This flow allows the usage of the accelerator for multiple algorithms with the highest possible throughput, which is the most intuitive way of using the MIMO accelerator.

4.2.2. Algorithm-First Design Flow

The algorithm-first design flow depends on the accelerator HDL parameters to optimize the hardware implementation for a certain algorithm. Fig. 4.2 shows a flow chart for the algorithm-first design flow.

If the application that will employ the MIMO accelerator hardware is known before the chip is implemented, the MIMO accelerator might not be reprogrammed after implementation. The designer can then start from the algorithm as shown in Fig. 4.2 by writing the high-level accelerator program based on the application in hand. The HDL parameters are then chosen based on this written program as it will be the only program running. A *parameter extraction*

tool can be used to detect which parameter values are suitable for a certain program. This tool will be introduced later in this chapter. This way, the accelerator hardware will be optimized in area, throughput, and power consumption for a particular application.

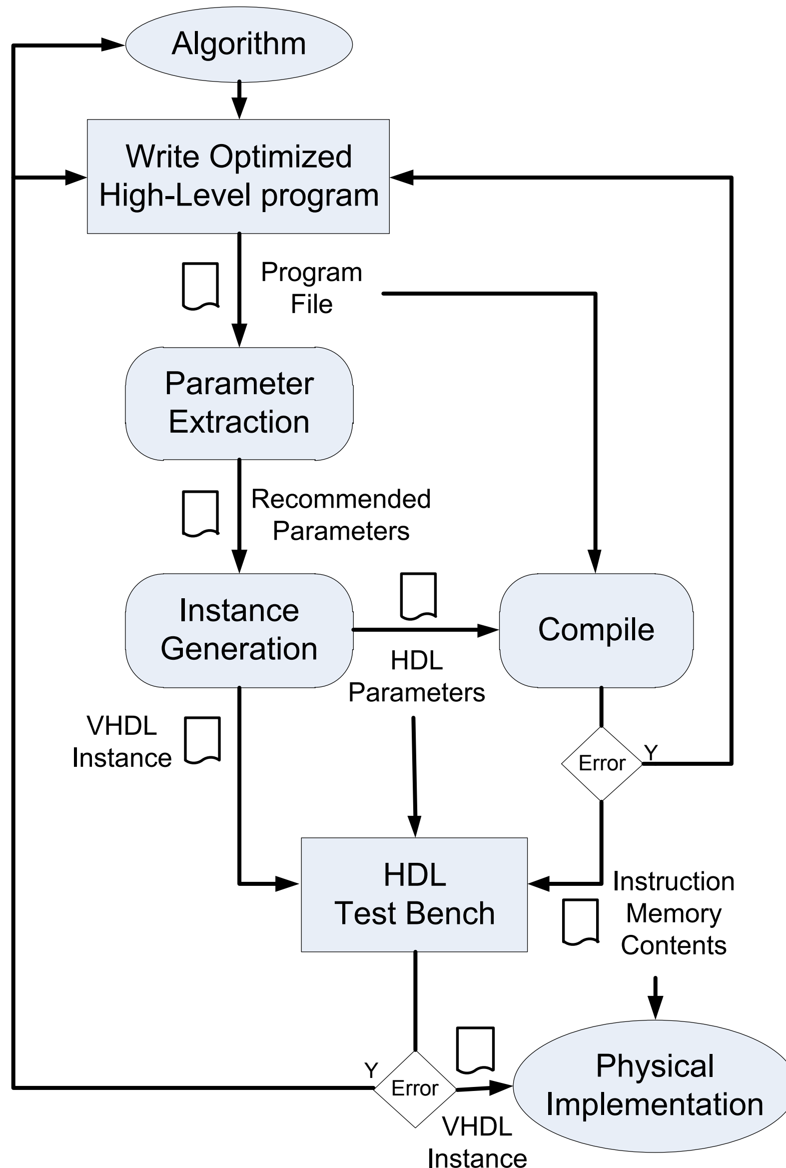

Fig. 4.2. Algorithm-first design flow to optimize the MIMO accelerator hardware for a specific application

The instance-generation tool takes the application-based HDL parameters and produces a VHDL description for the accelerator and a matched test bench. The generated instance and the test bench are used with the compiled program to test the program. The compilation output

(instruction memory contents) and the generated hardware description are used in the physical implementation to produce the final accelerator chip. The instruction memory here can be implemented as a ROM as it is not meant to be changed.

The final product of this flow is a MIMO accelerator that is optimized for a single application. But the produced chip is still re-programmable, if needed. The instruction memory might be overwritten with another algorithm.

4.3. MIMO Accelerator Programming

The MIMO accelerator is a processor that executes a pre-decoded programs stored in the instruction memory. Any accelerator user will face a very challenging task of writing the bit-level instructions, given the fact that these instructions control each and every part of the accelerator. A user will have to deeply study the accelerator hardware before writing programs that efficiently use the available hardware resources to achieve a certain task. This will be counter-productive as one of the accelerator objectives is to reduce the time needed to design a MIMO decoder. To simplify a programmer's task, an instruction set is developed for the accelerator, used in the design flows in the previous section, to cover the expected instructions that a user might need. This section covers this instruction set as well as the compiler interface, assuming N_{rx} (number of receive streams) of 4 and number of variable matrices per subcarrier (N_m) of 4 as well. Both N_{rx} and N_m are HDL parameters, as previously discussed.

4.3.1. Bit-Level Instruction

As previously mentioned in section 3.1 – and as shown in Fig. 4.3, an accelerator bit-level instruction consists of five main parts: flow control, processing-unit configuration, operand

selection, result destination, and write mask. Table 4.2 summarizes their usage showing that an instruction is 326 bits wide. This bit width of an instruction varies based on the HDL parameter values.

Flow Control 16b	Proc. Config. 15b	Operand Selection 132b	Result Destination 99b	Write Mask 64b

Fig. 4.3. Accelerator pre-decoded instruction structure

Table 4.2. The structure of a MIMO accelerator bit-level instruction

Field	Bit width	Description
Flow Control	16	Carries controller configuration bits. The relationship between the instruction counter and the subcarrier counter is controlled by this part of the instruction.
Processing Unit Configuration	15	Configures the four processing cores. This part is used to determine whether an addition or subtraction is used in the addition core. It also controls the dynamic scaling behavior for the outputs of both the multiplication and the reciprocal units. The applied unitary transformation from the rotation core is modified and configured by this part as well.
Operand Selection	132	Carries all the selection lines for the two-level multiplexers of the core-input switch. For each processing unit operand, 2 selection lines are used for the first level, and 4x16 selection lines are used for the second level to pick up one matrix element for each of the 16 elements of the operand. This gives the complete routing flexibility for the operands.

Field	Bit width	Description
Result Destination	99	Carries the selection lines for the two-level multiplexers of the memory-input switch. 2 bits are used to select which core delivers the result. Those two bits are also used to determine the passive cores whose inputs will be fixed while this particular instruction is executed. 1 bit is used to determine if phases of the rotation core will replace part of the outputs. And 2x16x3 bits are used to select one element of 2x4 outputs of the processing unit (3 bits selection) to be routed to each element of two matrices in the data memory (2x16 matrix elements). This allows complete flexibility in routing any element of the result to any memory location spanning two different matrices in the data memory.
Write Mask	64	All the variable matrices occupy one memory location for each subcarrier. This part of the instruction controls which element of the 4x16 elements will be overwritten by the result and which elements will keep their old values unchanged.

4.3.2. High-Level Instructions

Based on the bit-level structure of table 4.2, the higher-level instruction has the following structure: "*Result = Opearnd_1 Operator Operand_2 Pivot;*". The result and the two operands use a MATLAB-like structure for the ease of use. The variable matrices in the data memory have the fixed names A, B, C, and D. If more than four matrices are available, the naming continues

up to Z giving 24 as a maximum number of supported matrices by the instruction-set (P and Q have special use in the compiler as will be discussed later). Table 4.3 summarizes the various ways the result and the operands can be written.

Table 4.3. Styles for the result and the operands in a high-level instruction

Usage	Example	Allowed for Results	Allowed for Operands
Row Vector	A(2, :)	Yes	Yes
Column Vector	C(:, 3)	Yes	Yes
Matrix Diagonal	B(::)	Yes	Yes
Complete Matrix	D	Yes	Yes
Two Row Vectors	C(1:3, :) 1st and 3rd rows of C	Yes	Yes
Two Column Vectors	D(:, 2:3) 2nd and 3rd columns of D	Yes	Yes
Two Row Vectors Spanning Two Matrices	AB(2:2, :) 2nd row of A and 2nd row of B	Yes	No
Two Column Vectors Spanning Two Matrices	CD(:, 1:3) 1st column of C and 3rd column of D	Yes	No
Complex-Conjugate Modifier (~)	A(2,:)~	No	Yes
	D~	No	Yes

The high-level instruction also contains the operator, which obviously determines the operation to be performed on the two operands. Tables 4.4 to 4.6 list all the available operators and their use.

Table 4.4. Multiplication, addition, and reciprocal operators in the accelerator instruction set

Processing Core	Operator	Usage	Example
Multiplication	*	Vector-Vector Multiplication.	A(1, :) = B(2, :) * C(4, :);
		Vector-Matrix Multiplication.	A(:, 3) = C(2, :) * D;
		Matrix Multiplication.	C = A*B;
Addition	+/-	Single-Vector Addition/Subtraction.	A(1, :) = B(2, :) - C(4, :);
		Two-Vector Addition/Subtraction.	B(:, 1:4) = C(:, 2:3)~ + D(:, 3:4);
		Matrix Addition/Subtraction.	C = A + B~;
Reciprocal	./	Compute 1/X for the real parts of the second operand. The first operand is always replaced by one.	A(::) = 1 ./ D(::);

For the multiplication instructions, a vector-vector multiplication performs a dot product between the two vectors whether they are column-vectors, row-vectors, matrix-diagonals, or a combination. In this case, the four dot-product hardware blocks in the processing unit perform the same operation with the same operands giving a repeated number that will be stored in the

result vector. For the matrix multiplication, the compiler translates one high-level instruction into 4 bit-level instructions (or N_{rx} in general).

The addition and subtraction, in hardware, processes two vectors simultaneously. The compiler then translates a matrix addition/subtraction into 2 bit-level instructions (or $N_{rx}/2$ in general). When the compiler detects two consecutive single-vector addition or subtraction high-level instructions, it combines them in a single bit-level instruction. But if only one instruction is detected, it is repeated over the two parallel hardware blocks discarding one of the two results.

Table 4.5. Branching operators in the MIMO accelerator instruction set

Processing Core	Operator	Usage	Example
N/A	JMP	Unconditional branching. With no operands, the result is the absolute value of the branching address in the instruction memory.	8 = JMP;
		Conditional Branching. When operand 1 is present, this instruction checks the sign bit of the real part of first number of the last addition/subtraction result. If positive, the next executed instruction address is the result value. If negative, the next executed instruction address is the operand 1 value.	8 = 15 JMP;

Branching instructions are built with the same structure of regular instructions whether it is the conditional or unconditional branching as shown in table 4.5.

An odd part of the instruction is the pivot that is written between the second operator and the semi colon. This pivot is only used for some of the instructions that use the rotation unit, as explained in table 4.6. It is a number that determines which element of the two-operand vectors that will be used to calculate θ and ϕ using the super vectoring CORDIC. ϕ is the phase of the pivot element of the first operand vector. And $\theta = \tan^{-1}\frac{x}{y}$ where x is the amplitude of the pivot element of the second operand vector and y is the amplitude of the pivot element of the first operand vector. This is the arrangement needed to null an element using a complex Givens rotation. If the pivot is set to 0, the phases are picked up from the phase memory and divided by 2 (right shifted). And if the pivot is set to -1, the phases are picked up from the phase memory and divided by -2 (sign change and right shift).

Table 4.6. Rotation operators in the MIMO accelerator instruction set

Processing Core	Operator	Usage	Example
Rotation	@	Perform a complex Givens rotation $\begin{bmatrix} \cos\theta & \sin\theta e^{-j\phi} \\ -\sin\theta & \cos\theta\, e^{-j\phi} \end{bmatrix}$ for the two operand vectors based on operand 1, θ and ϕ are calculated from the two operands from the pivot elements.	A(1:2, :) = A(1, :) @ A(2, :) 1;
	^	Same as @ but uses the θ and ϕ from the previous instruction.	B(1:2, :) = B(1, :) ^ B(2, :);

Processing Core	Operator	Usage	Example
Rotation	#	Perform the unitary transformation $\begin{bmatrix} e^{-j\emptyset} & 0 \\ 0 & 1 \end{bmatrix}$ to eliminate the phase of a certain matrix element. ϕ is calculated from the first operand pivot element. There isn't a second operand for this operation.	A(1, :) = A(1, :) # 1;
	!	Same as # but uses ϕ of the previous instruction.	B(1, :) = B(1, :) !;
	p	Performs the same exact operation of # and stores the calculated ϕ in the phase memory.	A(1, :) = A(1, :) p 1;
	q	Performs the same exact operation of # and stores the calculated ϕ in the data memory to be used later for additional processing.	A(1, :) = A(1, :) q 1;
	P	Performs the same exact operation of @ and stores the calculated θ and ϕ in the phase memory.	A(1:2, :) = A(1, :) P A(2, :) 1;

Processing Core	Operator	Usage	Example
Rotation	Q	Performs the same exact operation of @ and stores the calculated θ and ϕ in the data memory to be used later for additional processing.	A(1:2, :) = A(1, :) Q A(2, :) 1;
	$	Perform a real Givens rotation $\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$ for the two operand vectors based on operand 1, θ is the phase memory output divided by 2.	B(:, 1:2) = B(:, 1) $ B(:,2);

The MIMO accelerator is not a stand-alone processor; it is a part of a bigger system in hardware. For interaction between the accelerator software and the top-level hardware, special no-operation instructions are added as shown in table 4.7. The NOPS (No-Operation-Stop) instruction is meant to save the unnecessary energy consumed after the program is done. Instead of using an infinite loop that keeps the complete accelerator running, the accelerator stops all operations and memory interactions waiting for the top-level to read the results and re-initialize the data memory. NOPI (No-Operation-Invert) is used to inform the hardware that this place of the program is reached. This allows the top-level hardware to start a side job in parallel to the accelerator at the correct timing.

Table 4.7. Special NOP instructions

Instruction	Usage
NOP	An actual No Operation instruction.
NOPS	Stops the operation, nothing is executed after the last stage of the previous instruction. It also de-activates a "work indicator" signal to the top level.
NOPI	No operation but inverts a signal to the top level indicating the execution of this instruction.

To build a bit-level instruction, the compiler translates the result of a high-level instruction into the "result destination" and the "write mask" fields of the bit-level instruction. The used operator determines a part of the "result destination" and controls which processing unit operands are updated and which are gated. A two-vector result is allowed to span two variable matrices in memory, as shown in Table 4.3. But due to the hardware restrictions, the only two-matrix access combinations allowed as a result are AB and CD. The compiler also translates the two operands into the "operand selection" field. And obviously the "processing-unit configuration" part of the instruction is extracted from the operator and the pivot. The complex-conjugate modifier appears as a part of the "processing-unit configuration" as well. It directly controls a complex-conjugate hardware modifier at the processing unit inputs.

To generate the remaining field, "flow control", the compiler uses the order of instructions. Regularly, each instruction is executed over all the subcarriers. That makes the regular operation is to always increment the subcarrier counter, and increment the instruction counter when the subcarrier counter resets. This flow can change based on the running instructions. For example,

if instruction number *i* is a rotation instruction that uses the phase from the previous instruction *i-1*, then both instructions *i-1* and *i* should be executed before incrementing the subcarrier counter. Then for every increment in the subcarrier counter, the instruction counter jumps between *i-1* and *i*. This kind of irregularity is called a *multi-cycle* operation. The compiler captures it from the order and type of instructions, and the hardware picks it up from the "flow control" field. The "flow control" also indicates if a certain instruction is a branching instruction. In this case, the "operand selection" field is replaced by the new value of the instruction counter, or two values if it is a conditional branching.

4.3.3. Compiler Interface

For a simple and a quick access, a GUI is created as an interface to the compiler with the addition of some useful features. Fig. 4.4 shows that the compiler GUI is divided into three main sections. The first section (Fig. 4.4a) is the code entry and compilation part. A user can load a program from an ASCII file or enter a new program in the text window and save it to disk. When the *compile button* is clicked, it first checks the syntax of the entered program. Part of the syntax checking is to guarantee that the hardware parameters are not violated. For example, if N_m and N_{rx} are both set to 4 in the hardware parameters, the compiler detects any vector or matrix dimension higher than 4 and any operand (or result) out of {A, B, C, D}, and the compiler then reports a parameter violation as a warning. After checking the syntax, the compiler starts the translation process from the high-level to bit-level instructions. The instruction memory contents are saved in ASCII to the *target file*.

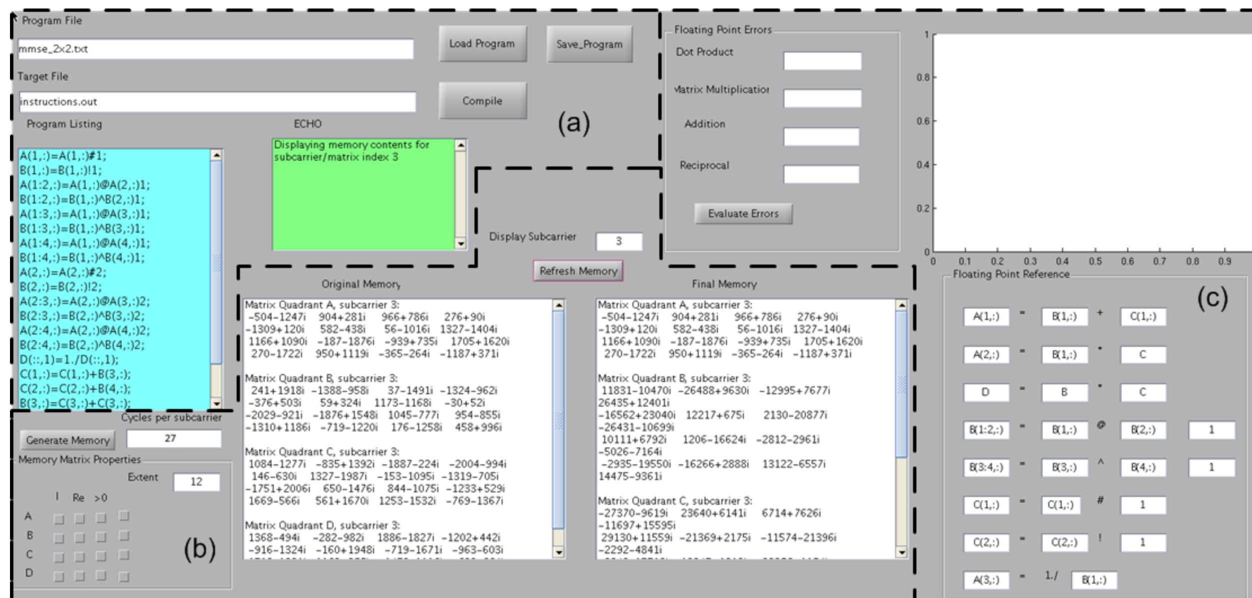

Fig. 4.4. MIMO accelerator compiler graphical user interface

The second section of the GUI (Fig. 4.4b) is for HDL simulation. The *generate memory button* is used to generate random contents for the data memory, which is displayed in the *original memory* display box for the chosen subcarrier. This generated data memory contents with the generated instruction memory are used as an input for a VHDL test bench for hardware simulation. The simulation results (the final values in the data memory after simulation) are displayed in the *final memory* box for the chosen subcarrier.

The third section of the compiler GUI (Fig. 4.4c) calculates the error between the accelerator results and a floating-point simulation results. It uses the generated data memory, but it doesn't use the generated instructions. A fixed set of instructions is used for this simulation to cover all the processing cores. The average error in the final memory is then displayed per processing core and plotted per subcarrier. This simulation is useful to check whether the error due to the fixed-point and the used precision is acceptable or not.

4.3.4. Example Program

As an example, Fig. 4.4 also shows a program, in the program entry section, that calculates the W matrix of an MMSE decoder, based on [16]. This program is repeated in Table 4.8 with instruction-by-instruction explanation.

The channel matrix $H_{2\times2}$ is used with the noise variance N_o to form the matrix $\begin{bmatrix} H_{2\times2} \\ \sqrt{N_o} I_{2\times2} \end{bmatrix}$, which is stored in the first two columns of the variable matrix A. And the $I_{4\times4}$ matrix is stored in the variable matrix B. Instructions 1 to 14 in table 4 perform the QR decomposition with the final $R_{4\times2}$ matrix stored in A. The $Q_{4\times4}$ matrix inverse ($Q^{-1} = Q^H$) is computed in B by applying the same unitary transformation of A on the I matrix in B. And let $Q_{4\times2} = \begin{bmatrix} Q_{1,2\times2} \\ Q_{2,2\times2} \end{bmatrix}$. The MMSE matrix is then computed as $W_{2\times2} = \frac{Q_2 \times Q_1^H}{\sqrt{N_o}}$ by the remaining instructions assuming an initialization of $D_{4\times4} = \sqrt{N_o} I_{4\times4}$. This W matrix is used by simple multiplication instructions to compute the estimated transmitted symbols ($\hat{x}_i[n] = Wy[n]$).

Table 4.8. Example program used for 2x2 MMSE MIMO decoding

Index	Instruction	Description
1	A(1,:)=A(1,:)#1;	Unitary transformation to eliminate the phase of A(1,1)
2	B(1,:)=B(1,:)!1;	Repeat the last operation on the I matrix
3	A(1:2,:)=A(1,:)@A(2,:)1;	Unitary transformation to null A(2,1)
4	B(1:2,:)=B(1,:)^B(2,:)1;	Repeat the last operation on the I matrix
5	A(1:3,:)=A(1,:)@A(3,:)1;	Unitary transformation to null A(3,1)
6	B(1:3,:)=B(1,:)^B(3,:)1;	Repeat the last operation on the I matrix

Index	Instruction	Description
7	A(1:4,:)=A(1,:)@A(4,:)1;	Unitary transformation to null A(4,1)
8	B(1:4,:)=B(1,:)^B(4,:)1;	Repeat the last operation on the I matrix
9	A(2,:)=A(2,:)#2;	Unitary transformation to eliminate the phase of A(2,2)
10	B(2,:)=B(2,:)!2;	Repeat the last operation on the I matrix
11	A(2:3,:)=A(2,:)@A(3,:)2;	Unitary transformation to null A(3,2)
12	B(2:3,:)=B(2,:)^B(3,:)2;	Repeat the last operation on the I matrix
13	A(2:4,:)=A(2,:)@A(4,:)2;	Unitary transformation to null A(4,2)
14	B(2:4,:)=B(2,:)^B(4,:)2;	Repeat the last operation on the I matrix
15	D(::)=1./D(::);	Compute $1/\sqrt{N_o}$ by inverting the diagonal of D
16	C(1,:)=C(1,:)+B(3,:);	Move Q_2 from the last two rows of B to the first two rows of C (C starts with zeros)
17	C(2,:)=C(2,:)+B(4,:);	
18	B(3,:)=C(3,:)+C(3,:);	Store zeros in the last two rows of B (C starts with zeros)
19	B(4,:)=C(4,:)+C(4,:);	
20	C=D*C~;	Computes $C = \frac{Q_2}{\sqrt{N_o}}$
21	A=B*C;	Computes $A = \frac{Q_2 \times Q_1^{*T}}{\sqrt{N_o}}$

4.4. Hardware Instance Generation

Processor-like hardware architecture accompanied with an easy-to-use compiler gives the MIMO accelerator a programmability aspect that allows a single hardware block to be capable of changing algorithms (programs) based on the updates in requirements. This section discusses the other aspect of the MIMO accelerator, which is the hardware configurability via the HDL

parameters. As discussed in section 4.1, The MIMO accelerator design in HDL is highly reconfigurable using a set of HDL parameters. An accelerator hardware instance is a VHDL description of the accelerator with the parameters set to specific values such as the prototype values shown in Table 4.1

To reconfigure the HDL with new parameter values, an accelerator user can directly change the HDL description. But to avoid involving a user with the hardware design, a simple interface GUI is designed for *instance generation*. As shown in Fig. 4.5, an accelerator user enters the HDL parameters in the left section of the instance generation GUI. Text boxes are used for the parameters with continuous values, such as N_{rx} and N_{sc}. Check boxes and option boxes are used for the binary parameters.

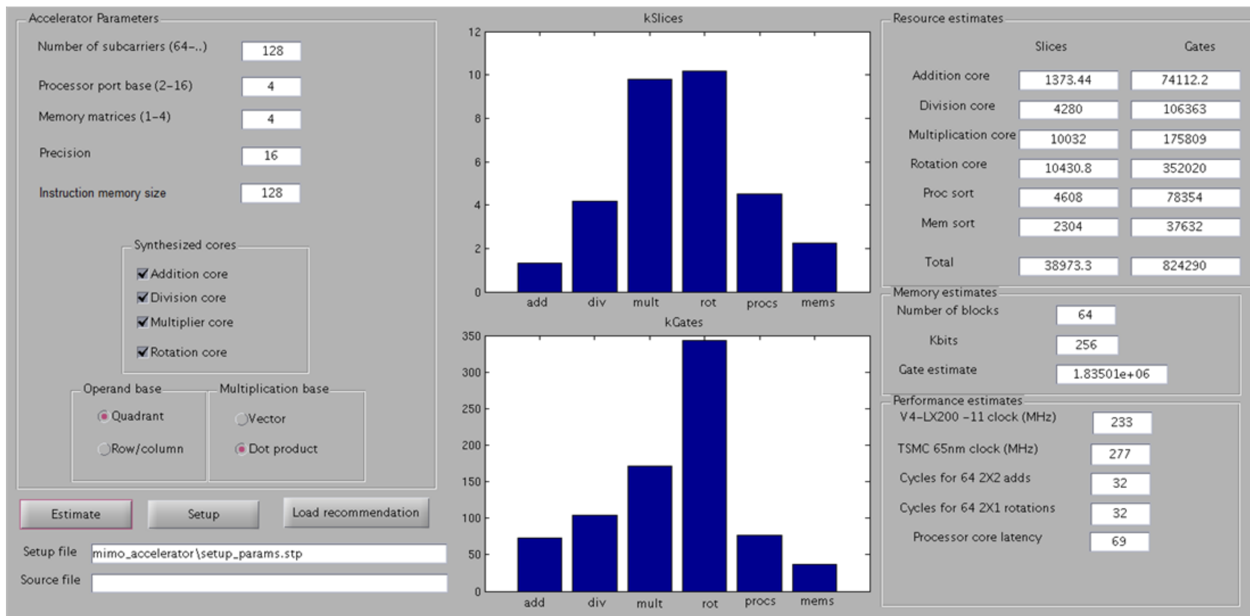

Fig. 4.5. MIMO accelerator instance generation graphical user interface

The instance generation GUI provides a user with an estimation of the expected hardware based on the entered parameters. The middle section of the GUI (Fig. 4.5) gives a plot for the individual areas of various building blocks of the accelerator, and the right section shows the corresponding numbers with estimation of the clock frequency and throughput. All estimations are provided for Xilinx V4 FPGA and for TSMC 65nm process based on a combination of a database of synthesis results, curves generated by interpolation and extrapolation, and empiric correction factors. The instance generation GUI provides the hardware estimates to be used for a quick comparison only; a full synthesis, whether for an FPGA flow or an ASIC flow, should be used to get final numbers.

The instance-generation GUI produces the VHDL of the MIMO accelerator with the required parameters. It also generates a setup file that contains those parameters. The compiler uses this setup file to check parameter-violations in a program. The VHDL test bench also uses the same setup file to simulate the generated hardware instance.

4.5. Parameter Extraction

If the programs that will run on the accelerator are known before the hardware generation process (algorithm-first design flow), a tool can give a recommendation for the parameter values based on those programs. The *parameter-extraction* GUI, as shown in Fig. 4.6, uses an ASCII list of programs as an input. It picks-up the required parameters to allow the execution of that list of programs. The recommendations of the parameter-extraction GUI can be loaded and used in the instance-generation GUI.

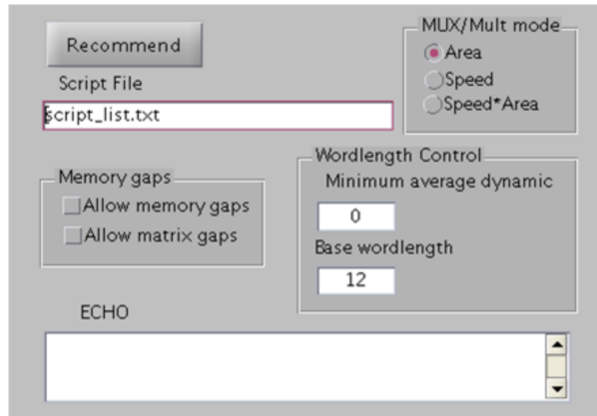

Fig. 4.6. Parameter extraction graphical user interface

In the parameter-extraction GUI, a user controls – through the "Allow memory gaps" check box – whether the number of variable matrices of the data memory will be based on the number of matrices used in the program list (no gaps), or it will be based on the actual used letter (allow gaps). For example, if the program list uses *D* as the only variable matrix, allowing memory gaps will recommend the number of matrices N_m to be 4. But a no-gaps setting will recommend N_m to be just 1. The same applies for the matrix dimensions (N_{rx}) through the "allow matrix gaps" check box.

The precision parameter *P* doesn't depend on the program list. The parameter-extraction GUI uses fixed-point simulation data for matrix multiplication, with dynamic scaling, to estimate the minimum possible precision for the user-provided SNR constraint. M_v/M_d parameters don't depend on the programs list either. Their values are recommended based on the area and latency estimation of the hardware.

5. MIMO Accelerator Prototype Chip

Based on the hardware-first design flow, a prototype MIMO accelerator chip is fabricated in 65nm IBM 10SF CMOS technology. This chip is meant to test the accelerator operation for various algorithms and programs and to compare its energy consumption with the dedicated ASIC implementations in the literature.

The chosen parameters for the prototype chip are shown in table 4.1, which is repeated here in table 5.1.

Table 5.1. MIMO accelerator HDL parameters and their values for the ASIC prototype

Parameter	Description	Prototype Value
N_{rx}	Number of data streams (vector size)	4
N_m	Number of data-memory variable matrices	4
N_{sc}	Number of OFDM subcarriers (data memory depth)	52
P	Fixed point precision per rail	16
N_{ins}	Maximum possible number of instruction (instruction memory depth)	256
T_a, T_r, T_m, T_v	Availability of individual processing units	1, 1, 1, 1
M_v/M_d	0/1: All processing units have the same number of outputs 1/0: All processing units have the same number of inputs	0/1
D_r/D_q	0/1: A complete data matrix can be routed to an operand 1/0: One vector only can be routed to an operand	0/1

This chapter starts with the test setup for the chip and then will go through the chip measurement results and their significance.

5.1. Test Setup and Procedure

To build a prototype chip for a custom and complex system like the MIMO accelerator, a complete test plan has to be in place by the design-freeze time for the tape-out. Fig. 5.1 shows a block diagram of the overall testing setup. A VHDL test bench is used for generating the accelerator test vectors. This means to generate the initial contents and the expected final contents of the data memory for a certain algorithm that is to be stored in the instruction memory.

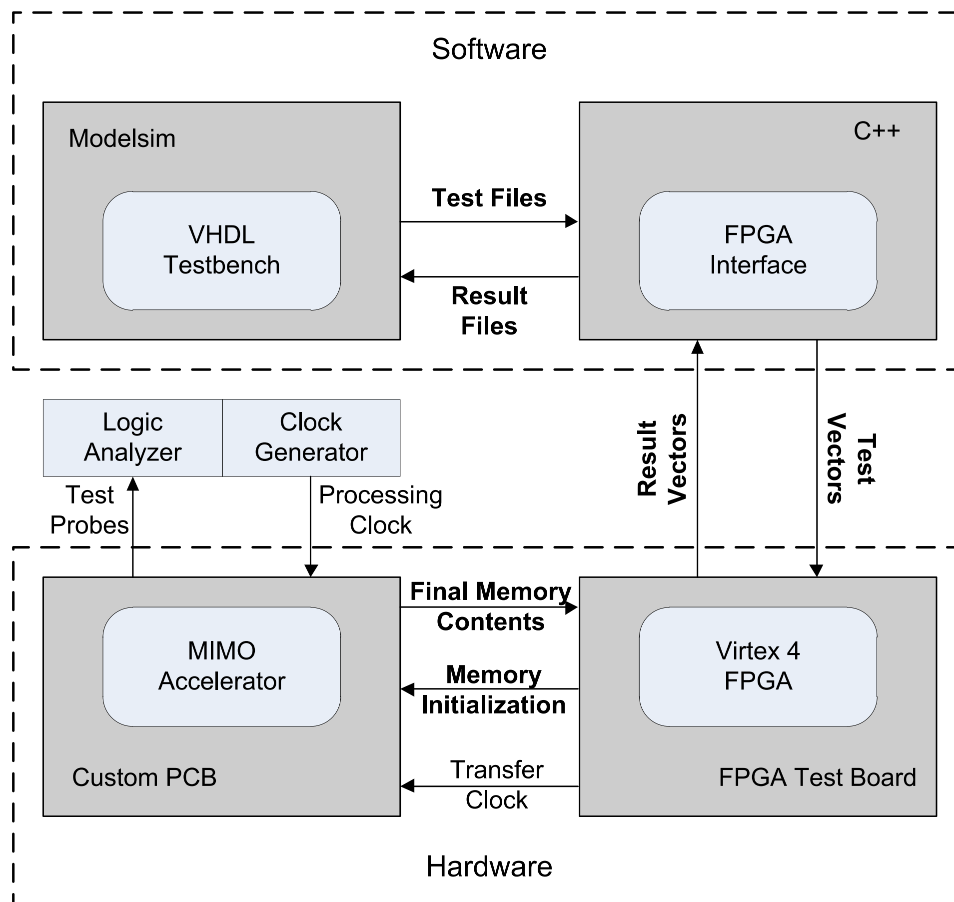

Fig. 5.1. Test setup for the MIMO accelerator chip

A Virtex-4 (4VLX160-11) FPGA on a Nallatech BenIO FPGA board is used to transfer the memory contents to and from the accelerator. The BenIO FPGA board is a daughter board

mounted on a Nallatech Bennuey board that is connected to the PCI bus of a PC. Custom software is written for the interaction between the VHDL test bench and the FPGA.

A custom printed circuit board (PCB) is designed and fabricated to connect the accelerator to the supply regulators, clock generator, logic analyzer probes, and the FPGA test signals. Fig. 5.2 shows a photo for the test setup in the lab, highlighting the built PCB, the FPGA board, and lab equipment.

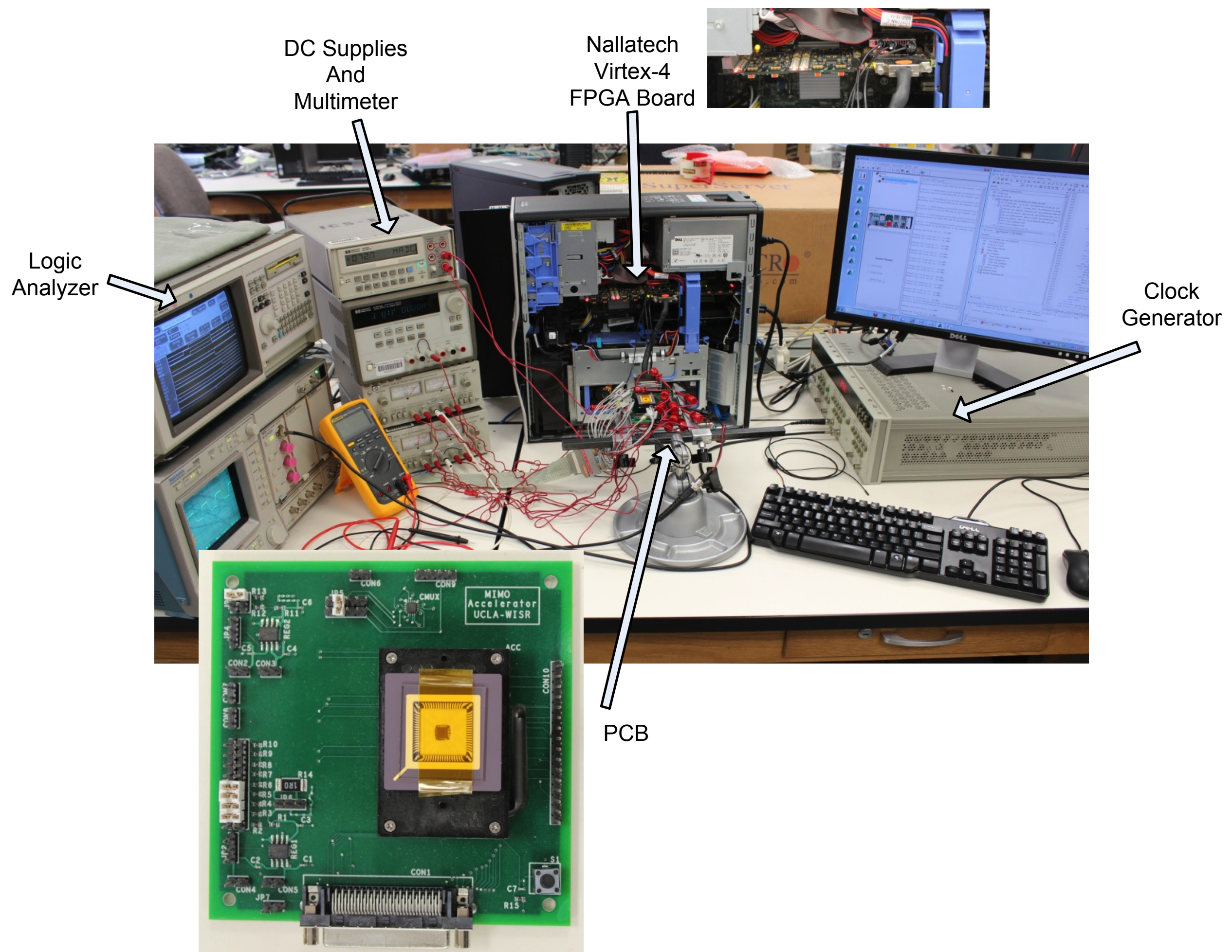

Fig. 5.2. Photo for the lab test setup in

An obvious problem facing the accelerator testing is that both the data and instruction memories have wide word lengths per location (2048 bits and 326 bits respectively). This huge number of bits can't be directly connected between the FPGA and the accelerator chip. To solve this problem, extra hardware blocks are designed and added on-chip only for testing as shown in Fig. 5.3. The on-chip test hardware acts as the top-level design for the MIMO accelerator. It controls the operation of the accelerator and arranges the data transfer between the accelerator and the FPGA.

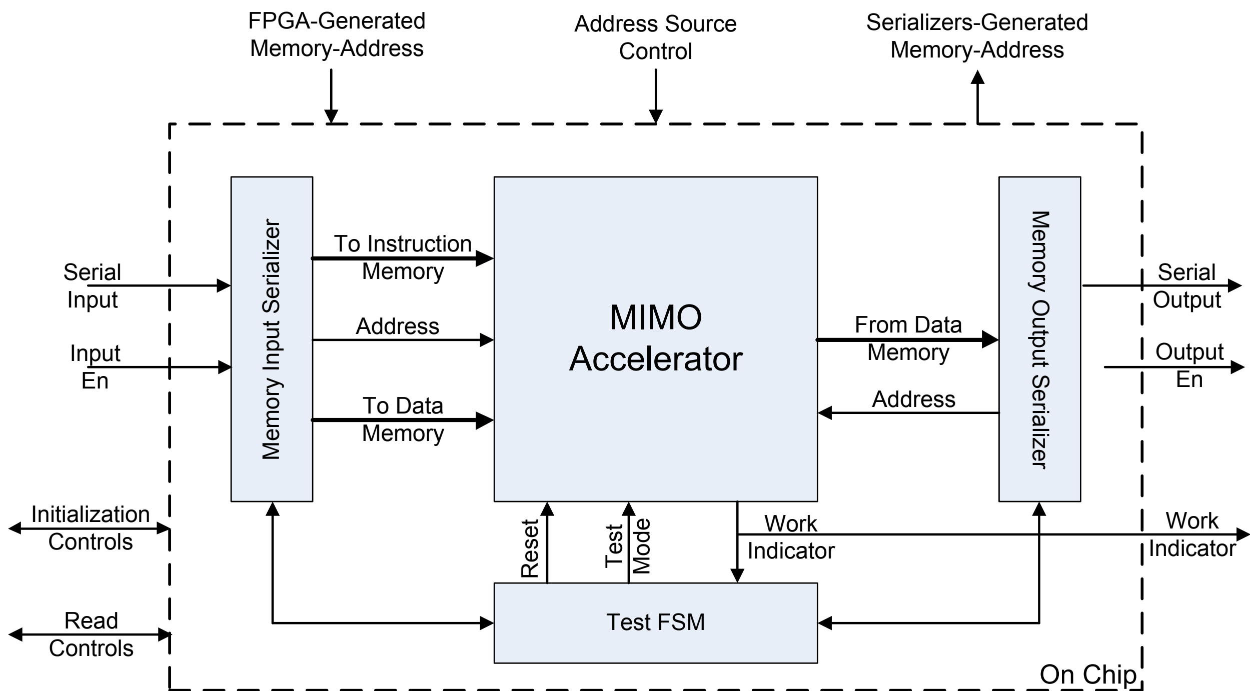

Fig. 5.3. On-chip test hardware blocks

The memory-input serializer, in Fig. 5.3, works as a serial-to-parallel converter from the FPGA side to the accelerator. The memory-output serializer does the same job in the other direction. On one side, the two serializers are connected to the accelerator through the top-level ports. And, on the other side, they are connected to the FPGA by two 8-bits data busses.

Moving into the details of the serializers, each one is a wide shift register – as shown in Fig. 5.4. For the output serializer, as an example, the output-data bus is always connected to the least significant eight bits of the shift register. One complete data memory location is read and stored in the shift register with a parallel load input. After receiving a ready-to-receive signal from the FPGA, the serializer performs an 8-bit right shift and gives an output enable signal. This shifting, which is based on the handshaking signals, is then repeated until the complete data location is transferred from the accelerator to the FPGA, and then a new memory location is loaded and transferred. The input serializer follows the same sequence of operation but in the other direction for both the data and instruction memories. The memory address used by the input and the output serializers can be generated internally using an address counter, the address in this case is given as a chip output to the FPGA tester. The input address can also be used to give the FPGA tester a full control on the accelerator and its on-chip testing hardware.

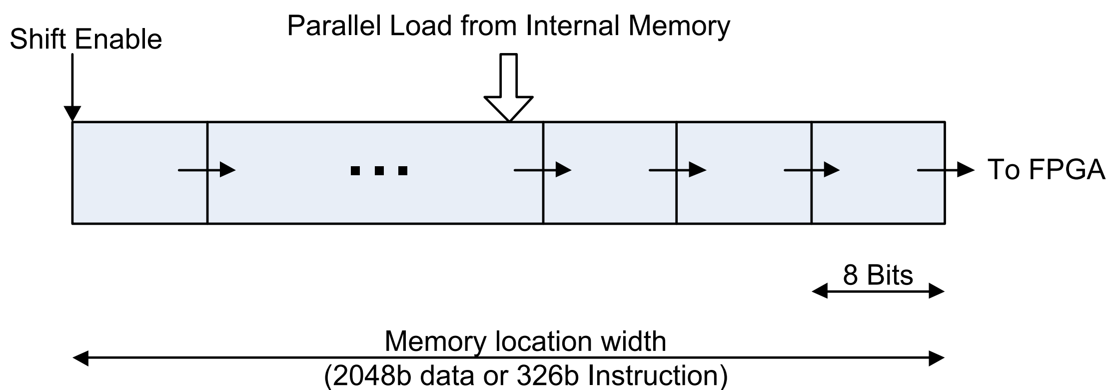

Fig. 5.4. On-chip parallel-to-serial and serial-to-parallel converters

In addition to the serializers, Fig. 5.3 also shows a FSM that is used to control the test procedure on chip. It acts as the top level control for the accelerator by generating the accelerator reset

signal and by controlling the memory read and write operations. This FSM also insures the synchronization between the chip and the FPGA for the time boundaries separating the initialization, accelerator run, and the results-read operations. The test FSM uses a work-indicator flag generated by the accelerator. This work indicator is asserted when the first instruction execution starts, and it is reset when a NOPS instruction is executed.

By creating this test setup, an extra effort is needed to guarantee correct data transfer between the FPGA and the accelerator chip despite the possible presence of a large clock skew between the two sides. A slow clock (20MHz) is generated and used by the FPGA tester and supplied to the accelerator chip during the data transfer times as its main clock of operation. During the accelerator run time, the actual test clock (fast clock – 166MHz) is supplied to the accelerator from a signal generator. This clock transition is made possible by a high speed MUX that is controlled by the FPGA tester. As shown in Fig. 5.5, the slow clock is supplied to the accelerator during the initialization time. After the final initialization data word is enabled on the data bus, the FPGA tester routes the test clock to the accelerator chip. When the work indicator is reset, the FPGA tester reroutes the slow clock back to the accelerator chip. This scheme guarantees a reliable link between the FPGA and the accelerator, and in the same time it allows the accelerator to run at full speed.

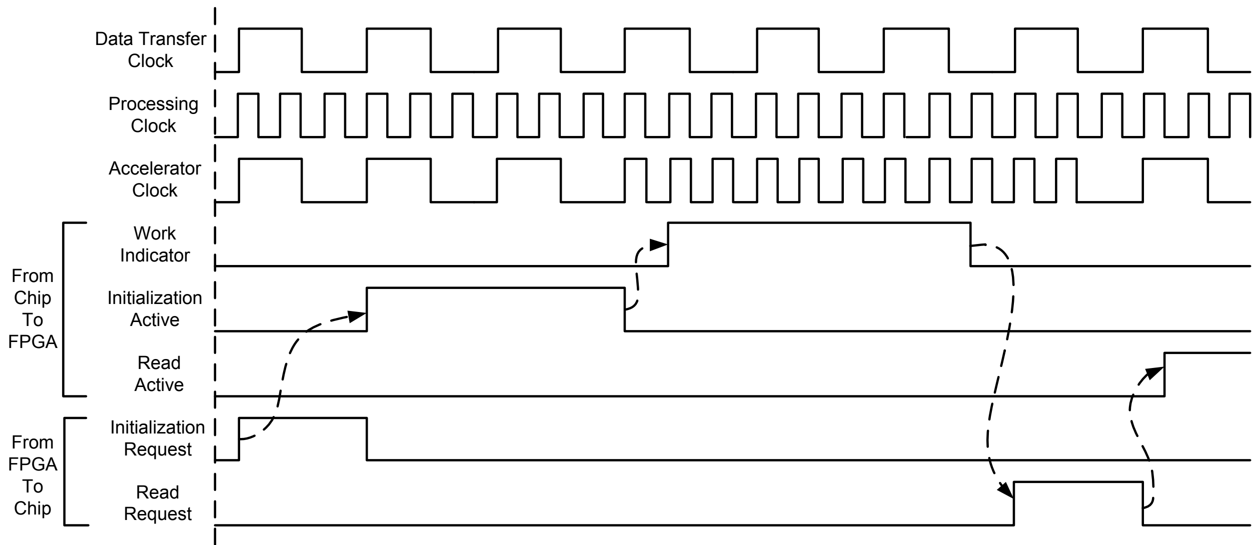

Fig. 5.5. Timing diagram for the interaction between the test signals to control the clock multiplexing

5.2. Measurement Results

The MIMO accelerator ASIC was fabricated in IBM 65nm regular CMOS (10SF) technology. The complete die area is 7.56mm^2, and the chip core area (excluding the IO pads) is 6.05mm^2. Excluding the memory, the accelerator is 2.48mm^2, which is equivalent to 469k gates. Fig. 5.6 shows the MIMO accelerator chip micrograph.

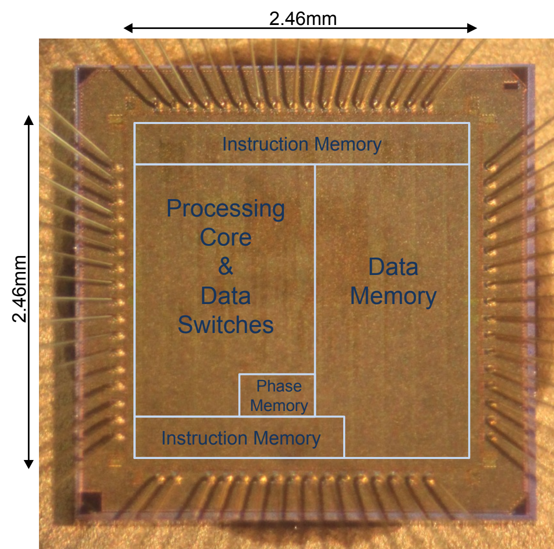

Fig. 5.6. MIMO Accelerator chip micrograph

Big part of the chip area is used for memory, whether it is the data memory or the instruction memory. Table 5.2 shows the area breakdown for the main blocks of the MIMO accelerator. The data memory is 44% of the accelerator due to the fact of using 4 antennas and 4 data variables. This number of variable matrices might not be necessary for all programs, but – as a prototype chip – this wide memory is added to open new possibilities in test programs, such as 8x4 QR factorization and complete 4x4 MMSE MIMO decoding.

Table 5.2. The percentage of the main blocks area to the complete chip core area

Block	Processing Core	Controller	Memory Switches	Phase Memory	Instruction Memory	Data Memory	Serializers & Test
Area%	36%	0.05%	2.3%	0.7%	14.9%	44%	1.05%

The minimum possible clock period (in measurements) is 6ns (166MHz). This clock frequency number is not useful without exploring its significance on the accelerator applications. Table 5.3 shows the number of clock cycles needed to finish an algorithm per subcarrier. The numbers are based on the measured throughput; table 5.3 assumes a continuous accelerator operation and doesn’t count the initial latency in the reported number of clock cycles. The table also translates the number of clocks to an actual time, based on the maximum clock frequency of 166MHz. The MMSE decoding mentioned in Table III is based on [16].

Table 5.3. Time needed to finish an algorithm running on the accelerator prototype

Algorithm	Clocks Per Subcarrier	Time Per Subcarrier	Comments
4x4 QR Decomposition	20	120ns	Both Q and R available
8x4 QR Decomposition	40	240ns	Both Q and R available
4x4 SVD	52	313ns	
Complete 2x2 MMSE	35	210ns	W-matrix computation including the QRD
Complete 4x4 MMSE	58	348ns	W-matrix computation including the QRD
One symbol decode in 2x2	0.5	3ns	Vector-Matrix multiplication
One symbol decode in 4x4	1	6ns	Vector-Matrix multiplication

To complete the analysis, Table 5.4 shows the time requirements for two MIMO-OFDM communication standards: 802.11n [1] as a simple standard in terms of processing speed requirements, and LTE-A [4] as a more demanding standard. For 802.11n, a re-sync is done every packet, which is assumed to be a long packet of 2ms (the packet length is variable). On the other side, the LTE-A requires a re-sync every 5 sub-frames, and a sub-frame is composed of 2 slots, which gives a re-sync interval of 5ms. Assuming MMSE is used for MIMO decoding on the accelerator, calculating the W matrix is required for every re-sync, and decoding (multiply the received vector by W) is required with a rate related to the bandwidth (subcarrier spacing and number of subcarriers). Equation (5.1) is used for calculating the minimum required clock frequency to calculate W (F1) under the assumption that a receiver is allowed 10% of the re-sync time to calculate W – which is dependent on the system memory. And equation (5.2) is used for calculating the minimum required clock frequency to decode (F2).

$$F1 = \frac{N_{sc} \times N_W}{T_{sync} \times 0.1} \qquad (5.1)$$

$$F2 = F_{sc} \times N_{sc} \times N_D \qquad (5.2)$$

Where N_{sc} is the number of subcarriers, N_w is the number of clock cycles the accelerator uses to compute W for a subcarrier, T_{sync} is the re-sync time interval, F_{sc} is the subcarrier spacing, and N_D is the number of clock cycles the accelerator uses to decode a subcarrier.

Table 5.4. Summary for a subset of relevant 802.11n and LTE-A parameters

Subcarrier Spacing	OFDM Subcarriers	MIMO Operation	Slot/Packet Time Interval	Re-sync Time Interval	Min Clock to Calculate W (F1)	Min Clock to Decode (F2)
802.11n						
312.5kHz	64	4x4	2ms	2ms	19MHz	20MHz
LTE-A						
15kHz	1024	4x4	0.5ms	5ms	119MHz	16MHz

Equations (1) and (2) are considered the worst case analysis as the subcarriers are not all used for data, and the 10% time limit in equation (5.1) is an aggressive limit. One more thing to notice is that the number of subcarriers only affects the data memory size of the accelerator. Which means the assumption that the clock speed will stay the same in hardware for a different number of subcarriers is a reasonable assumption.

Tables 5.3 and 5.4 with the rest of the analysis shows that the 166MHz is actually significantly better than what is the required by the modern communication standards.

The average power consumption of the accelerator is 300.9mW at 166MHz clock and 1V supply. Fig. 5.7 shows the measured power consumption for different clock frequencies and the energy consumed per clock cycle. The plot shows that as the clock frequency increases the power increases linearly. But the energy consumed in a clock cycle follows a 1/X trend. This is due to

the leakage power, which is not affected by the change in frequency. This leads to the fact that the higher the clock frequency is, the more energy efficient the accelerator will be.

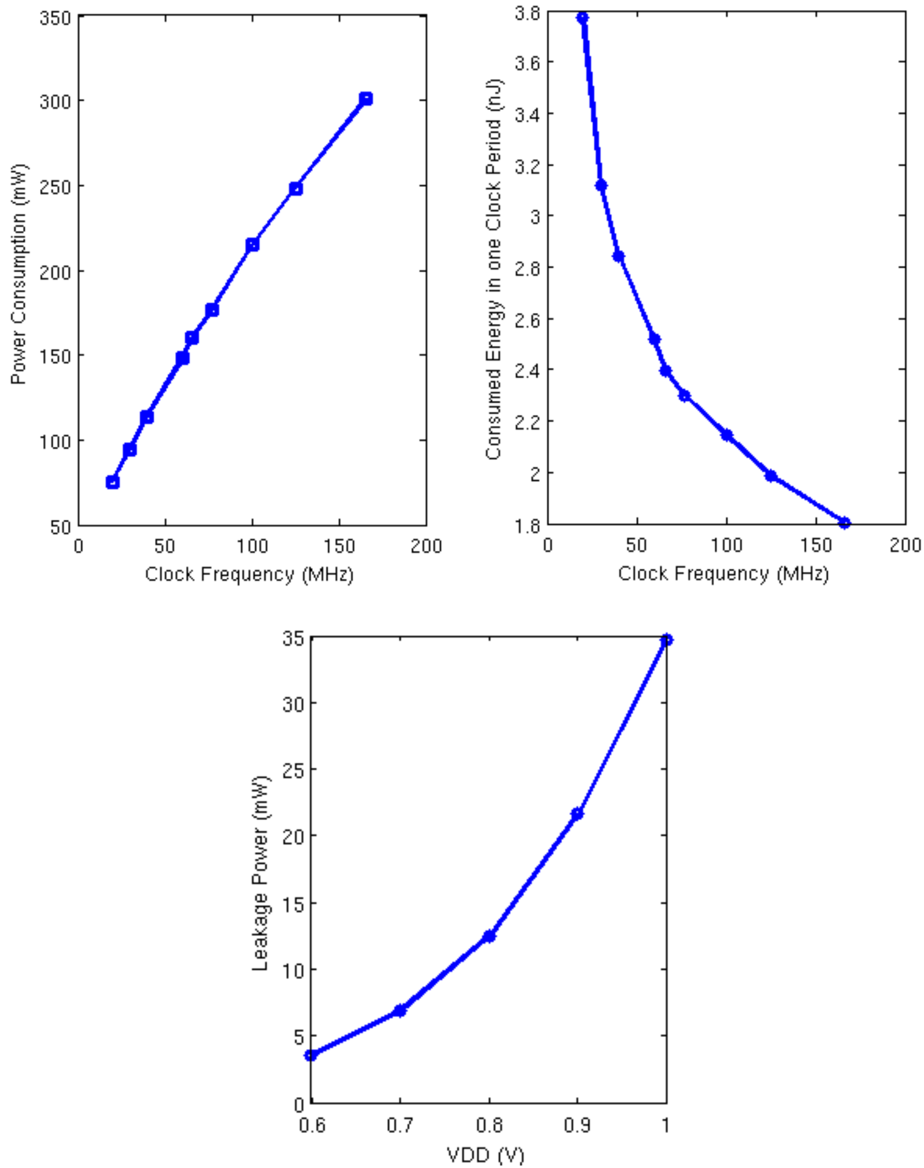

Fig. 5.7: Power consumption and energy per one clock cycle versus clock frequency,

And leakage power versus supply voltage

The measured leakage power consumption falls exponentially by reducing the supply voltage, as Fig. 5.7 also shows. This leads to the fact that if an application requires a low throughput, a

slower clock can be used with a reduction in the supply voltage to reduce the leakage power consumption, and hence the overall power consumption.

To guarantee correct operation in an actual receiver, a floating-point MATLAB model for the 802.11n is used to test the accelerator. The inputs to the MIMO decoder (channel matrices and received symbols) are quantized and stored as input test vectors for the MIMO accelerator. The results of the accelerator are then fed back to the MATLAB model to complete the receiver processing. Fig. 5.9 and Fig. 5.10 show a comparison between the accelerator and floating-point MMSE MIMO decoding for 16-QAM and 64-QAM. 100 packets are simulated for each point with a payload of 12k bits under channel D of the Wi-Fi channel models.

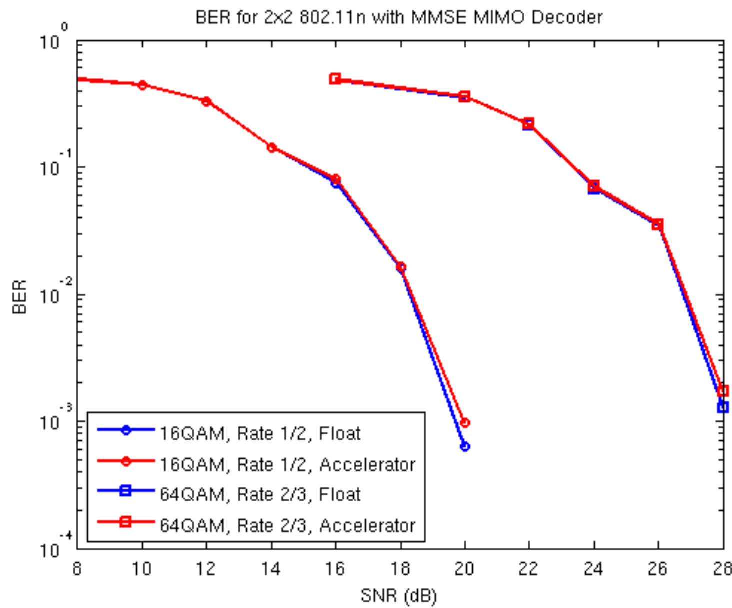

Fig. 5.9. BER comparison between MMSE on the accelerator and floating point simulation

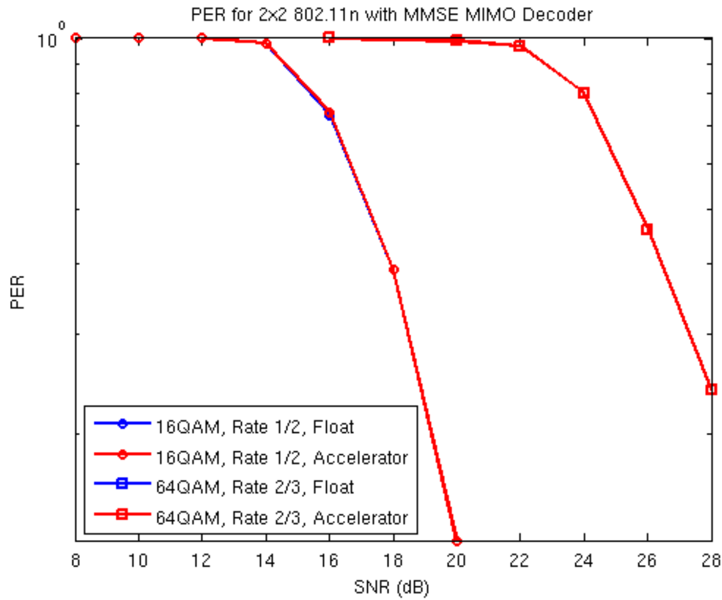

Fig. 5.9. PER comparison between MMSE on the accelerator and floating point simulation

Table 5.5 compares the accelerator with specialized ASIC designs for different MIMO processing algorithms. In the 4x4 QRD category, the accelerator is compared to two designs reported in [21]. The accelerator penalty in energy consumption is only 10% compared to the two dedicated designs. In [28], a 4x4 QRD ASIC is reported without the power numbers, but the comparison highlights the higher throughput of the accelerator.

In [21], the two QRD designs are re-used for implementing 4x4 SVD. The accelerator consumes half the energy of the two designs. This shows the benefit of the accelerator programmability. Also in [22], a 4x4 SVD is reported based on iterative methods to track the Eigen values of a channel matrix. To reach the first SVD before tracking, it takes 500 clock cycles. The accelerator is 30% better in energy consumption when compared to this first computation of the SVD.

Table 5.5. Comparison to other ASIC designs

Algorithm	Reference	Clk Freq	Average Power	Area	Tech.	Computation Time*/Tech Scaled	Energy Consumption/ Tech Scaled	Normalized Tech Scaled Energy
4x4 QRD	[21]a	133MHz	155mW	$0.41mm^2$	0.18µm	1.92µs / 450ns	297.6nJ / 33.2nJ	0.922
	[21]b	272MHz	105mW	$0.37mm^2$	0.18µm	2.82µs / 662ns	296.1nJ / 33nJ	0.916
	[28]	162MHz	NA**	$1mm^2$	0.18µm	642ns / 150ns	NA**	NA**
	Accelerator	166MHz	300.9mW	$2.48mm^2$	65nm	120ns / 120ns	36nJ / 36nJ	1
4x4 SVD	[21]a	133MHz	160mW	$0.41mm^2$	0.18µm	11.57µs / 2.7µs	1.85µJ / 206.3nJ	2.20
	[21]b	272MHz	106mW	$0.37mm^2$	0.18µm	15.83µs / 3.7µs	1.67µJ / 187nJ	1.99
	[22]	100MHz	34mW	$3.5mm^2$	90nm	5µs / 2.6µs	170nJ / 122.8nJ	1.31
	Accelerator	166MHz	300.9mW	$2.48mm^2$	65nm	313ns	94nJ / 94nJ	1
2x2 MMSE	[30]	58MHz	360mW	2.3M Gates	0.18µm	NA**	0.69nJ ◆	0.381 ◆
	Accelerator	166MHz	300.9mW	469K Gates	65nm	210ns / 210ns	1.81nJ ◆	1 ◆
4x4 MMSE	[29] ◆◆	400MHz	NA**	90K Gates	65nm	505ns / 505ns	NA**	NA**
	Accelerator	166MHz	300.9mW	469K Gates	65nm	348ns / 348ns	104.7nJ / 104.7nJ	1

* The required time to complete the running algorithm once.

** Not reported and not enough data to compute.

◆ Energy consumption in one clock period.

◆◆ Synthesis results only.

In [30], a 2x2 MMSE is reported as part of a receiver. The throughput of the MIMO decoder is not included. This comparison is only for the energy in one clock cycle, which maps to the power consumption. We included this comparison to show that the accelerator can be compared to completely different dedicated designs.

Finally, Table V compares the accelerator to the programmable implementation in [29] based on the reported synthesis results. As the power numbers are not included in [29], the comparison shows the throughput improvement of the accelerator despite the lower clock frequency.

6. Conclusions

This work introduced the MIMO accelerator as a programmable and energy efficient hardware block for MIMO decoding tasks in an OFDM system. Based on a processor-like structure that includes data and instruction memories, the accelerator design is optimized to reduce the overall energy consumption of a running algorithm. The accelerator processing-core, with its parallel and reconfigurable processing elements, is powerful enough to perform any required algorithm in less than 64 instructions.

The hardware implementation of the accelerator overcame many challenges such as the instruction memory access, data memory arrangement, system integration, and processing unit power-saving. This hardware design led to a prototype chip in 65nm CMOS that can be compared to various designs such as QR decomposition, SVD, MMSE linear decoders, and other algorithms. The chip testing has proven that the accelerator can – in some cases – be twice as energy efficient as dedicated ASIC designs, and 9% worse in energy consumption in the worst comparison.

The accelerator high-level programming language and design tools were introduced with their graphical user interfaces. Two design flows that utilize the accelerator to its maximum value were introduced based on those tools.

7. Future Work

This work may be extended, in future research, in the following directions:

- The MIMO accelerator can be integrated in a complete receiver in hardware to test the overall performance of a system that depends on the accelerator.

- The prototype chip of the accelerator is powerful enough to run multiple algorithms. Another chip might be fabricated based on the algorithm-first design flow. This might lead to better energy efficiency for this particular algorithm.

- Based on the idea of building an accelerator for MIMO decoding in OFDM systems, the same idea of a specialized processor can be applied to other parts of wireless transceivers in the direction of creating a software-defined radio.

References

[1] "IEEE Standard for Information technology-- Local and metropolitan area networks-- Specific requirements-- Part 11: Wireless LAN Medium Access Control (MAC)and Physical Layer (PHY) Specifications Amendment 5: Enhancements for Higher Throughput," *IEEE Std 802.11n-2009 (Amendment to IEEE Std 802.11-2007 as amended by IEEE Std 802.11k-2008, IEEE Std 802.11r-2008, IEEE Std 802.11y-2008, and IEEE Std 802.11w-2009)*, Oct. 29, 2009.

[2] "IEEE Standard for Local and metropolitan area networks Part 16: Air Interface for Broadband Wireless Access Systems Amendment 3: Advanced Air Interface," *IEEE Std 802.16m-2011(Amendment to IEEE Std 802.16-2009)* , May 5, 2011.

[3] 3GPP TS 36.201 V8.3.0 (2009-03).

[4] “LTE-Advanced Physical Layer”, REV-090003r1, IMT-Advanced Evaluation Workshop 17 – 18 December, 2009, Beijing.

[5] Andrea Goldsmith, “Wireless Communications”, Cambridge University Press, 2005.

[6] S. M. Alamouti, "A simple transmit diversity technique for wireless communications ," IEEE Journal on Selected Areas in Communications, 1998.

[7] W. Gabran, B. Daneshrad, "Hardware and Physical Layer Adaptation for a Power Constrained MIMO OFDM System," IEEE International Conference on Communications (ICC), 2011.

[8] A. J. Paulraj, D. A. Gore, R. U. Nabar, H. Bolcskei, "An overview of MIMO communications - a key to gigabit wireless," Proceedings of the IEEE , 2004.

[9] David Tse, Pramod Viswanath, “Fundamentals of Wireless Communcation”, Cambridge University Press, 2005.

[10] I. LaRoche, S. Roy, "An efficient regular matrix inversion circuit architecture for MIMO processing," IEEE International Symposium on Circuits and Systems, 2006.

[11] Lei Ma, K. Dickson, J. McAllister, J. McCanny, "QR Decomposition-Based Matrix Inversion for High Performance Embedded MIMO Receivers," IEEE Transactions on Signal Processing, 2011.

[12] M. Myllyla, J. H. Hintikka, J. R. Cavallaro, M. Juntti, M. Limingoja, A. Byman, "Complexity Analysis of MMSE Detector Architectures for MIMO OFDM Systems," Conference Record of the Thirty-Ninth Asilomar Conference on Signals, Systems and Computers, 2005.

[13] F. Echman, V. Owall, "A scalable pipelined complex valued matrix inversion architecture," IEEE International Symposium on Circuits and Systems, 2005

[14] M. Karkooti, J. R. Cavallaro, C. Dick, "FPGA Implementation of Matrix Inversion Using QRD-RLS Algorithm," Conference Record of the Thirty-Ninth Asilomar Conference on Signals, Systems and Computers, 2005.

[15] M. Ylinen, A. Burian, J. Takala, "Updating matrix inverse in fixed-point representation: Direct versus iterative methods," in IEEE International Symposium on System-on-Chip, Tampere, Finland, November 2003.

[16] H. S. Kim, W. Zhu, J. Bhatia, K. Mohammed, A. Shah, B. Daneshrad, "A practical, hardware friendly MMSE detector for MIMO-OFDM based systems," EURASIP Journal on Advances in Signal Processing, vol. 2008.

[17] B. Hassibi, H. Vikalo, "On the sphere-decoding algorithm I. Expected complexity," IEEE Transactions on Signal Processing, 2005.

[18] R. Shariat-Yazdi, T. Kwasniewski, "A multi-mode sphere detector architecture for WLAN applications," IEEE International SOC Conference, 2008.

[19] Chia-Hsiang Yang, D. Markovic, "A Flexible DSP Architecture for MIMO Sphere Decoding," IEEE Transactions on Circuits and Systems I, 2009.

[20] Zhan Guo, P. Nilsson, "Algorithm and implementation of the K-best sphere decoding for MIMO detection," IEEE Journal on Selected Areas in Communications, 2006.

[21] C. Studer, P. Blosch, P. Friedli, A. Burg, "Matrix Decomposition Architecture for MIMO Systems: Design and Implementation Trade-offs," Conference Record of the Forty-First Asilomar Conference on Signals, Systems and Computers, 2007.

[22] D. Markovic, R. W. Brodersen, B. Nikolic, "A 70GOPS, 34mW Multi-Carrier MIMO Chip in $3.5mm^2$," Symposium on VLSI Circuits, 2006.

[23] P. W. Wolniansky, G. J. Foschini, G. D. Golden, R. A. Valenzuela, "V-BLAST: an architecture for realizing very high data rates over the rich-scattering wireless channel," International Symposium on Signals, Systems, and Electronics, 1998.

[24] Wong Kwan Wai, Chi-Ying Tsui, R. S. Cheng, "A low complexity architecture of the V-BLAST system," Wireless Communications and Networking Confernce, 2000.

[25] D. Wubben, R. Bohnke, V. Kuhn, K. D. Kammeyer, "MMSE extension of V-BLAST based on sorted QR decomposition," IEEE 58th Vehicular Technology Conference, 2003.

[26] G. H. Golub and C. F. Van Loan, "Matrix Computations", Johns Hopkins University Press, Baltimore, Md, USA, 3rd edition, 1996.

[27] A. Jain, G. Staino, P. Corsonello, “Quad-port memory blocks in radiation-tolerant FPGAs: an application for image processing systems,” 2nd International Conference On Emerging Trends in Engineering and Technology, 2009.

[28] P. Luethi, C. Studer, S. Duetsch, E. Zgraggen, H. Kaeslin, N. Felber, W. Fichtner, "Gram-Schmidt-based QR decomposition for MIMO detection: VLSI implementation and comparison," IEEE Asia Pacific Conference on Circuits and Systems, 2008.

[29] J. Eilert, Di Wu, Dake Liu, "Implementation of a programmable linear MMSE detector for MIMO-OFDM," IEEE International Conference on Acoustics, Speech and Signal Processing, 2008.

[30] J. Wang, “A recursive least-squares ASIC for broadband 8 x 8 multiple-input multiple-output wireless communications,” Ph.D. dissertation, Henry Samueli School on Engineering and Applied Science, University of California in Los Angeles, Los Angeles, CA, 2005.