Verification and FPGA implementation of a floating point SIMD processor for MIMO processing

Examensarbete utfört i Datorteknik vid Tekniska högskolan i Linköping av

Sajid Hussain

LiTH-ISY-EX--10/4379--SE

Linköping 2010
Verification and FPGA implementation of a floating
point SIMD processor for MIMO processing

Examensarbete utfört i Datorteknik
vid Tekniska högskolan i Linköping
av

Sajid Hussain

LiTH-ISY-EX--10/4379--SE

Handledare: Andreas Ehliar
ISY, Linköpings universitet

Examinator: Andreas Ehliar
ISY, Linköpings universitet

Linköping, 9 December, 2010
Sammanfattning
Abstract

The rapidly increasing capabilities of digital electronics have increased the demand of Software Defined Radio (SDR), which were not possible in the special purpose hardware. These enhanced capabilities come at the cost of time due to complex operations involved in multi-antenna wireless communications, one of those operations is complex matrix inversion.

This thesis presents the verification and FPGA implementation of a SIMD processor, which was developed at Computer Engineering division of Linköping university, Sweden. This SIMD processor was designed specifically for performing complex matrix inversion in an efficient way, but it can also be reused for other operations. The processor is fully verified using all the possible combinations of instructions.

An optimized firmware for this processor is implemented for efficiently inverting 4×4 matrices. Due to large number of subtractions involved in direct analytical approach, it losses stability for 4×4 matrices. Instead of this, a blockwise subdivision is used, in which 4×4 matrix is subdivided into four 2×2 matrices. Based on these 2×2 matrices, the inverse of 4×4 matrix is computed using the direct analytical approach and some other computations.

Finally, the SIMD processor is integrated with Senior processor (a control processor) and synthesized on Xilinx, Virtex-4 FPGA. After this, the performance of the proposed architecture is evaluated. A firmware is implemented for the Senior which uploads and downloads data/program into the SIMD unit using both I/O and DMA.

Nyckelord
Keywords

SDR, SIMD, Xilinx, FPGA, DMA
Abstract

The rapidly increasing capabilities of digital electronics have increased the demand of Software Defined Radio (SDR), which were not possible in the special purpose hardware. These enhanced capabilities come at the cost of time due to complex operations involved in multi-antenna wireless communications, one of those operations is complex matrix inversion.

This thesis presents the verification and FPGA implementation of a SIMD processor, which was developed at Computer Engineering division of Linköping university, Sweden. This SIMD processor was designed specifically for performing complex matrix inversion in an efficient way, but it can also be reused for other operations. The processor is fully verified using all the possible combinations of instructions.

An optimized firmware for this processor is implemented for efficiently inverting $4 \times 4$ matrices. Due to large number of subtractions involved in direct analytical approach, it losses stability for $4 \times 4$ matrices. Instead of this, a blockwise subdivision is used, in which $4 \times 4$ matrix is subdivided into four $2 \times 2$ matrices. Based on these $2 \times 2$ matrices, the inverse of $4 \times 4$ matrix is computed using the direct analytical approach and some other computations.

Finally, the SIMD processor is integrated with Senior processor (a control processor) and synthesized on Xilinx, Virtex-4 FPGA. After this, the performance of the proposed architecture is evaluated. A firmware is implemented for the Senior which uploads and downloads data/program into the SIMD unit using both I/O and DMA.

Sammanfattning

Den snabbt ökande prestandan hos digital elektronik har ökat behovet av Software Defined Radio (SDR), vilket inte var möjligt med tidigare hårdvara. Den ökade förmåga kommer till priset av tidsåtgång, till följd av komplexa procedurer i samband med trådlös kommunikation med flera antenner, en av dessa procedurer är komplex matrisinvertering.

Denna avhandling presenterar verifiering och FPGA implementering hos en SIMD processor, vilken har blivit utvecklad vid institutionen för datorteknik, Linköpings universitet, Sverige. Denna SIMD processor blev specifikt designad för att
vi

Genomföra komplex matrisinvertering på ett effektivt sätt, men kan också användas för andra tillämpningar. Processorn har testats och verifierats för alla möjliga kombinationer av instruktioner.

En optimerad firmware för denna processor är implementerad för att effektivt invertera 4×4 matriser. På grund av att ett stort antal subtraktioner är inblandade i ett direkt analytiskt angreppssätt, så förlorar den stabilitet för 4×4 matriser. Istället används en stegvis indelning i underavdelningar, där 4×4 matrisen delas in i fyra 2×2 matriser. Baserat på dessa 2×2 matriser beräknas inversen av 4×4 matrisen med hjälp av ett direkt analytiskt angreppssätt samt andra beräkningar.

Slutligen, SIMD processorn är integrerad i en huvudprocessor och körs på Xilinx, Virtex-4 FPGA. Efter detta utvärderas prestandan hos den föreslagna arkitekturen. Firmware implementeras hos huvudprocessorn som laddar upp och ned data/program till SIMD enheten genom I/O samt DMA.
Acknowledgments

First, I would like to thank my supervisor and examiner Andreas Ehliar for his technical guidance, patience, and ever helping attitude. I am also grateful to Johan Eilert for helping me remotely, being patient and for prompt replies to all my silly emails. Johan, thank you very much for making this work successful.

My friends Ilyas Iqbal, Imran Hakam, Mati Ullah, Umar Farooq, Syed Ahmed Aamir, Ahmed Salim and those not listed here, I say profound thank for all kind of help and giving a nice company during my stay in Sweden.

Finally, my loving Ammi, Abbu and brothers for encouraging, unconditional support and always being there for me when I needed them. Thank you.

Sajid Hussain
Linköping, 2010
# Contents

1 Introduction  
   1.1 Objective .................................................. 1  
   1.2 Report Outline .......................................... 2  

2 SIMD Processor  
   2.1 Overview .................................................. 5  
   2.2 Pipeline Architecture ................................... 7  
   2.3 Module Description ....................................... 8  
      2.3.1 The SIMD I/O interface .............................. 8  
      2.3.2 MAC .................................................... 9  
      2.3.3 Register and Accumulator file ...................... 10  
      2.3.4 Special purpose register ........................... 10  
      2.3.5 Control registers .................................. 10  
   2.4 Memory ..................................................... 10  
      2.4.1 Program memory ..................................... 10  
      2.4.2 Data memory ......................................... 11  
      2.4.3 Stack .................................................. 11  
   2.5 Addressing Modes ......................................... 11  
      2.5.1 Relative addressing ................................ 11  
      2.5.2 Absolute addressing ................................ 11  
   2.6 Numerical Representation ................................ 12  
   2.7 Tools ....................................................... 12  
      2.7.1 Assembler ............................................. 13  
      2.7.2 Parser ................................................ 13  
      2.7.3 Simulator ............................................ 13  

3 Verification ........................................................ 17  
   3.1 Instruction Format ....................................... 17  
      3.1.1 Instruction encoding ................................ 17  
   3.2 Pipeline Delays ........................................... 19  
      3.2.1 Stalling or manual NOPs .............................. 19  
   3.3 Instruction Verification ................................... 21  
      3.3.1 Addition operation .................................. 21  
      3.3.2 Multiply operation ................................. 23
List of Figures

1.1 Block diagram of an 'Ideal' Software Defined Radio. . . . . . . 1

2.1 The SIMD unit execute states, E1, E2 and E3. . . . . . . . 6
2.2 The SIMD block unit. . . . . . . . . . . . . . . . . . . . . . . 8
2.3 Multiply-accumulate. . . . . . . . . . . . . . . . . . . . . . . 9
2.4 Register and accumulator file. . . . . . . . . . . . . . . . . 9

3.1 SIMD instruction format. . . . . . . . . . . . . . . . . . . . . . 18
3.2 Addition datapath. . . . . . . . . . . . . . . . . . . . . . . . 22
3.3 Multiply datapath. . . . . . . . . . . . . . . . . . . . . . . . 23
3.4 Multiply-accumulate datapath. . . . . . . . . . . . . . . . . . 25

4.1 MIMO communications. . . . . . . . . . . . . . . . . . . . . . 29
4.2 Memory utilization. . . . . . . . . . . . . . . . . . . . . . . . 33
4.3 Memory layout. . . . . . . . . . . . . . . . . . . . . . . . . . . 33

5.1 Overview of the peripheral interface. . . . . . . . . . . . . . . . 40
5.2 Senior and SIMD integration. . . . . . . . . . . . . . . . . . . . . 41
5.3 RS232 interface with the system. . . . . . . . . . . . . . . . . . . . 42
5.4 Virtex-4 evaluation board. . . . . . . . . . . . . . . . . . . . . . 43
5.5 ChipScope system setup. . . . . . . . . . . . . . . . . . . . . . . 48
<table>
<thead>
<tr>
<th>Abbreviations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMD</td>
<td>Single Instruction, Multiple Data</td>
</tr>
<tr>
<td>SDR</td>
<td>Software Defined Radio</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>OFDM</td>
<td>Orthogonal Frequency Division Multiplexing</td>
</tr>
<tr>
<td>LTE</td>
<td>Long Term Evolution</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>MIMO</td>
<td>Multiple Input, Multiple Output</td>
</tr>
<tr>
<td>I/O</td>
<td>Input, Output</td>
</tr>
<tr>
<td>MAC</td>
<td>Multiply-Accumulate</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Language</td>
</tr>
<tr>
<td>ISE</td>
<td>Integrated System Environment</td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier Transform</td>
</tr>
<tr>
<td>VLIW</td>
<td>Very Long Instruction Word</td>
</tr>
<tr>
<td>NOP</td>
<td>No Operation</td>
</tr>
<tr>
<td>SGR</td>
<td>Squared Givens Rotations</td>
</tr>
<tr>
<td>SPI</td>
<td>Serial Peripheral Interface</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>NGC</td>
<td>Native Generic Circuit</td>
</tr>
<tr>
<td>NGD</td>
<td>Native Generic Database</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>UCF</td>
<td>User Constraints File</td>
</tr>
<tr>
<td>CLB</td>
<td>Combinational Logic Block</td>
</tr>
<tr>
<td>NCD</td>
<td>Native Circuit Description</td>
</tr>
<tr>
<td>XST</td>
<td>Xilinx Synthesis Technology</td>
</tr>
<tr>
<td>ILA</td>
<td>Integrated Logic Analyzer</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The software defined radio (SDR) has been an essential part of recent research in radio developments. The idea behind the SDR is to fully configure the radio by the software as it can be used as a common platform. The software radio can easily be re-configured as upgrades of standard arrive or it can be customized according to specific requirements [1].

1.1 Objective

In SDR, most of the complex signal handling required in communications transmitters and receivers are done in the digital style. An analog to digital converter chip connected to an antenna can be considered as a fundamental form of SDR receiver. All the filtering and signal detection can take place in the digital domain, perhaps in an ordinary personal computer or embedded computing devices [2].

![Figure 1.1. Block diagram of an 'Ideal' Software Defined Radio.](image)

As number of users is increasing, quality of service (QoS) might be decreased gradually. It has been said that more intelligent air-interface schemes are required to efficiently utilize the radio spectrum and to mitigate the interference by users. The situation becomes worse when fading plays an unfavorable role. The link reliability, and gain diversity are also an essential requirement to adopt efficient utilization of bandwidth. In OFDM based radio system multiple users can share
resources in a way to achieve high spectral efficiency and peak to average power ratio [3].

In an OFDM air-interface there is training data transmitted first and then real data. The training data is used by the receiver to compute an estimate of how the radio channel distorts the received data and how the data is received by the different antennas. There are several ways to compute channel characteristics from training data, one of those ways is “Matrix Inversion” which is the most computational and time taking operation of OFDM.

In order to speed-up the system so that receiver has to store less data in its buffer and it may have valid information about channel for next coming data packets. The real data is typically transmitted immediately after the training data since otherwise the channel estimate computed from the training data will not be useful since the channel is continuously changing and will have changed.

In some OFDM based systems (LTE for example) there is training data interleaved with the real data so that the receiver can continuously update its channel estimate. These computations must also finish as quickly as possible so that the receiver can benefit from the updated channel estimate as soon as possible.

In order to mitigate this problem, an idea was proposed by some PhD students at the division of Computer Engineering at Linköping university. The idea is to outsource the matrix inversion to an external FPGA/ASIC using some efficient algorithm. For this purpose, a SIMD processor was developed but not verified by Johan Eilert (a former PhD student). This processor is not only specific to perform matrix inversion but can also be reused for other operations like complex multiplication, filtering, correlation and FFT/IFFT.

The purpose of this thesis work is to verify the instruction set of the SIMD unit and implement the algorithm for a 4x4 matrix inversion. As the SIMD unit is a target device, there is a need of master device which will configure the SIMD unit. The Senior processor (a local DSP processor of Computer Engineering division of Linköping university, Sweden) is integrated with the SIMD unit as a master device. Finally, the SIMD unit and the Senior are synthesized and performance of the proposed architecture is evaluated.

### 1.2 Report Outline

General information about the SIMD unit and its architecture is given in chapter 2. It also describes the development tools used during implementation, verification and synthesis.

Chapter 3 describes, how the instruction set of the SIMD unit is verified, bugs found during verification and how the instructions automatically stall whenever
needed.

The pros and cons of different matrix inversions, and how the blockwise matrix inversion is used using direct analytical approach are discussed in chapter 4.

Chapter 5 illustrates how the SIMD unit is integrated with the Senior processor and synthesized. It also summarizes the problems faced during synthesis and how they were debugged.

Chapter 6 shows the results obtained after synthesis. It also evaluates the algorithm proposed for the matrix inversion.

Chapter 7 describes the conclusions drawn from the thesis work.

Chapter 8 contains the list of tasks which can be done in future for improving the system.
Chapter 2

SIMD Processor

The SIMD processor is designed to meet the demands of efficient complex matrix inversion for MIMO software defined radios. But, it is designed in such a way, that it can also accommodate heterogeneous applications by loading their programs. So, it can be reused for many other applications [4]. As was mentioned in the first chapter that Johan Eilert designed this processor, but it was not verified. This chapter describes the hardware architecture of the SIMD unit.

2.1 Overview

The SIMD unit contains two separate datapaths for operations. One is left side datapath and other is right side datapath. They both work independently with their own instructions. One instruction consists of two parts; one part is for the left side datapath and other part is for the right side datapath.

The processor has separate data and program memory, both left and right side datapaths share data and program memory. The size of data and program memory is $40 \times 256$ bits and $32 \times 256$ bits respectively. All general purpose registers and accumulators can be accessed in parallel, and their native width is 40 bits.

The processor implements four parallel ways. All four parallel ways use the same program memory, which is common. At one time, the same instruction goes to all four ways. On the other hand, each way has its own data memory. The above mentioned size of data memory is for one way. Because of four parallel ways, it has four times wider data memory and registers, and produces four results in each operation. The total data memory size for all the four ways is $160 \times 256$ bits.

This SIMD unit deals with $4 \times 4$ antenna matrices. In order to compute all four matrices simultaneously, there are four identical datapaths; each datapath computes one matrix. There is no communication at all between the ways and they are identical with the exception of the reciprocal unit which is shared. Fig: 2.1 illustrates the architecture of the execute states of the SIMD unit [5].
Figure 2.1. The SIMD unit execute states, E1, E2 and E3.
2.2 Pipeline Architecture

The SIMD unit has different pipelines for the left and the right side datapath. Table: 2.1 shows the pipeline stages for the left and the right side datapath with further description in Table: 2.2.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Left side</th>
<th>Right side</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IF</td>
<td>IF</td>
</tr>
<tr>
<td>2</td>
<td>ID</td>
<td>ID</td>
</tr>
<tr>
<td>3</td>
<td>OF</td>
<td>OF/AG</td>
</tr>
<tr>
<td>4</td>
<td>E1</td>
<td>EX/MEM</td>
</tr>
<tr>
<td>5</td>
<td>E2</td>
<td>WB</td>
</tr>
<tr>
<td>6</td>
<td>E3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>WB</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1. Pipeline specification.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Instruction fetch</td>
</tr>
<tr>
<td>ID</td>
<td>Instruction decode</td>
</tr>
<tr>
<td>OF</td>
<td>Operand fetch</td>
</tr>
<tr>
<td>E1</td>
<td>Execution stage 1</td>
</tr>
<tr>
<td>E2</td>
<td>Execution stage 2</td>
</tr>
<tr>
<td>E3</td>
<td>Execution stage 3</td>
</tr>
<tr>
<td>WB</td>
<td>Write back to register file</td>
</tr>
<tr>
<td>OF/AG</td>
<td>Operand fetch/Address computation</td>
</tr>
<tr>
<td>EX/MEM</td>
<td>Execute/Memory access</td>
</tr>
</tbody>
</table>

Table 2.2. Explanation of pipeline stages.

On the left side, there are seven pipeline stages: IF, ID, OF, E1, E2, E3, WB. E1/2/3 is execute. E1 is multiplication, E2 and E3 is addition and accumulation, respectively. E1 and E2 together form a complex multiplication, E3 is complex addition. The execute stages can be viewed graphically in Fig: 2.1 [5].

The right pipeline is shorter: IF, ID, OF/AG, EX/MEM, WB. Here, AG is address generation and MEM is memory access.

2.3 Module Description

The following section describes briefly important modules and tools used in designing/verifications and their issues.
2.3.1 The SIMD I/O interface

The SIMD unit is a special purpose computational processor. It gets some input data and gives the results after some computation. The completion of computation is usually indicated by an interrupt signal as shown in Fig: 2.2. It has quite simple interface with the external environment.

![Figure 2.2. The SIMD block unit.](image)

Table: 2.3 shows the I/O signals descriptions. The address bus is 6 bits wide, and it can address to 64 locations. But, the data memory is 256 entries deep. In case of absolute addressing, it works well; since absolute addressing can only access 0-63 locations, whereas for relative addressing, the SIMD unit uses data memory pointer with this address for accessing entries beyond 64th location. In data bus, 0-15 bits are used for transferring data and 16th bit is used as a strobe signal.

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>in</td>
<td>System clock</td>
</tr>
<tr>
<td>reset</td>
<td>in</td>
<td>System reset</td>
</tr>
<tr>
<td>addr_in[5:0]</td>
<td>in</td>
<td>Address bus</td>
</tr>
<tr>
<td>data_in[15:0]</td>
<td>in</td>
<td>Input data bus</td>
</tr>
<tr>
<td>data_in[16]</td>
<td>in</td>
<td>Input data strobe</td>
</tr>
<tr>
<td>data_out[15:0]</td>
<td>out</td>
<td>Output data bus</td>
</tr>
<tr>
<td>data_out[16]</td>
<td>out</td>
<td>Output data strobe</td>
</tr>
<tr>
<td>interrupt</td>
<td>out</td>
<td>Interrupt signal</td>
</tr>
</tbody>
</table>

Table 2.3. The SIMD I/O interface.

2.3.2 MAC

All internal computations are done in 20 bits floating point numbers. The left side datapath of this processor includes one Floating-Point Complex Multiply-Accumulate (FPCMAC) unit. This FPCMAC is made up of one multiplier and two
adders, which are shared when there is need to perform separate multiplications or additions. Fig: 2.3 shows the layout of the MAC unit [4].

![Figure 2.3. Multiply-accumulate.](image)

### 2.3.3 Register and Accumulator file

There are 32 general purpose and 4 accumulator registers. All these registers are of 40 bits, 20 bits for storing real part and 20 bits for imaginary part. There are two ports for accessing the register and accumulator file. For some instructions, it is required to access two registers simultaneously. In that case, both ports are used by selecting multiplexers at the output. For example, \texttt{ADD acc,~reg1,~reg2} instruction accesses two registers at a time. It can access \texttt{reg1} and \texttt{reg2} by setting \texttt{of\_rf0\_mxc} and \texttt{of\_rf1\_mxc} mux select signals respectively. Fig: 2.4 shows the structure of the register and accumulator file [6].

![Figure 2.4. Register and accumulator file.](image)
2.3.4 Special purpose register

The SIMD unit has a special purpose register for taking reciprocal of operands. A write of some complex value to a special purpose register will invert the real number and returns zeros for the imaginary part after 10 clock cycles. This special purpose register is also of 40 bits in width.

2.3.5 Control registers

There are 12 control registers and each register is 16 bits wide. Each register maps to an address and it can only be accessed through the SIMD interface. The control registers are located in different modules of the SIMD unit. Table 2.4 shows the list of control registers, their addresses and functions.

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>0x09</td>
<td>Current PC, also PM write address</td>
</tr>
<tr>
<td>Program end</td>
<td>0x0a</td>
<td>Program end address</td>
</tr>
<tr>
<td>Loop start</td>
<td>0x0b</td>
<td>Loop start address (pc reload value)</td>
</tr>
<tr>
<td>Loop end</td>
<td>0x0c</td>
<td>Loop end address</td>
</tr>
<tr>
<td>Loop iteration</td>
<td>0x0d</td>
<td>Number of remaining iterations (pc reloads)</td>
</tr>
<tr>
<td>Data memory pointer</td>
<td>0x0e</td>
<td>Current DM pointer for program</td>
</tr>
<tr>
<td>Data memory step</td>
<td>0x10</td>
<td>Set DM step for program (used each reload)</td>
</tr>
<tr>
<td>Insn write port</td>
<td>0x11</td>
<td>Write high, low word of insn, then inc the pc</td>
</tr>
<tr>
<td>DMA pointer</td>
<td>0x12</td>
<td>DMA read/write pointer</td>
</tr>
<tr>
<td>DMA write port</td>
<td>0x13</td>
<td>Write 16b or 20b data into the DM</td>
</tr>
<tr>
<td>DMA read count</td>
<td>0x14</td>
<td>Set number of lines to read</td>
</tr>
<tr>
<td>Control</td>
<td>0x15</td>
<td>bit 0: Start executing at PC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 1: Enable INT when program ends</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 2: INT currently signalled, write 1 to ack</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 3: 0:16b-&gt;20b conversion, 1:16b raw mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit 4: DMA was started and is running</td>
</tr>
</tbody>
</table>

Table 2.4. Control registers specification.

2.4 Memory

2.4.1 Program memory

The SIMD unit fetches a new instruction each time from the program memory. This program memory is 32 bits wide and up to 256 entries deep. It can only be accessed by the instruction fetch unit; it cannot be used for storing data or constants.
2.5 Addressing Modes

2.4.2 Data memory

The data memory is used for storing input data and run time data. It contains garbage values after reset. It is 40 bits wide and 256 entries in depth.

2.4.3 Stack

The main intention of designing the SIMD unit was to perform the complex matrix inversion. For this purpose, an efficient algorithm was already suggested by the designer of the SIMD unit. The processor was designed based on that algorithm; in which there was no need to use a stack. This was the reason of not implementing the stack in the SIMD unit. It can be modified to add subroutines instructions and a stack if desired for other applications.

2.5 Addressing Modes

There are two addressing modes of the SIMD unit, one is relative addressing and other is absolute addressing.

2.5.1 Relative addressing

The relative addressing gives access to the data that should be processed in each iteration. This can be achieved by setting the data memory pointer and the step size register to a suitable values prior to running the program.

By relative addressing, the data memory can be accessed from 0-255. The relative addressing can only access data_ptr_i + offset where data_ptr_i has enough bits to reach all addresses and offset is from -32 to +31. The 6 input address bits are interpreted as a signed offset. At one time, it can access maximum 64 locations with the same data memory pointer value. For accessing locations beyond 64, the data memory pointer has to be updated by adding the step register to it.

Since, the SIMD unit was developed specifically for the complex matrix inversion. The matrix inversion algorithm is supposed to run on a number of matrices in a loop, where each iteration accesses the next matrix in memory. For this, one has to use the relative addressing and update data_mem_ptr to point to the next matrix after each iteration.

2.5.2 Absolute addressing

The absolute addressing can be used to access scratch variables or constants that must be stored in the data memory that are reused in every iteration. This addressing mode can only access addresses 0 to 63.
There was no need of scratch variables or constants in the matrix inversion code, that is why; the absolute addressing has not been used at all, but it can be used for other purposes.

### 2.6 Numerical Representation

It is difficult to choose the best numerical representation for baseband processing. Floating point representation is preferred for applications having higher dynamic range like radars and echo cancellers. On the other hand, fixed point representation is more suitable for area sensitive applications like mobile handsets [7]. Matrix inversion algorithms are more sensitive to finite length effects; it was difficult to choose the suitable representation. Some applications use fixed point while some adopt floating point representation [4].

In order to choose the best numerical representation, the designer of the SIMD unit implemented and synthesized a datapath on both FPGA and ASIC technologies to find the best trade off among different numerical representations. Based on those results, 20 bits (14 bits mantissa and 6 bits exponent) floating point representation brought the best receiver performance in that simulation [4].

IEEE 754 standard is used for the floating point representation [8]. Below is an expression used for conversion from binary to decimal number in the floating point arithmetic.

$$(-1)^{signbit} \times (1 + fraction) \times 2^{exponent-bias}$$

For example, 1 100001 0110110011001 is a 20 bits floating point number in binary. Bit 19 (the leftmost bit) represents the sign of the number. If this bit is 1, the number is negative, if it is 0, the number is positive; in this case it is 1, so the number is negative. Bits 13-18 (the next 6 bits) are the exponent, and it is 33. The bias is 31 for this case which is the maximum positive sign exponent. Bits 0-12 (on the right) give the fraction. Fractional part can be calculated as:

$$0.0110110011001 = 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} + 1 \times 2^{-6} + ...$$

The equivalent decimal floating point number is:

$$(-1)^1 \times (1 + 0.424926758) \times 2^{33-31} = -5.699707$$

### 2.7 Tools

A set of tools were implemented to be used during the implementation and verification of the SIMD unit. The functional simulation of the RTL is performed by Mentor Graphics ModelSim. Integrated System Environment (ISE) tool provided by Xilinx is used for synthesis, implementation and area estimation.
Assembler and instruction set simulator, which are used for instruction verification and matrix inversion firmware were already developed at the division of Computer Engineering at Linköping university, Sweden. Some features in the simulator were missing in the beginning like absolute and relative data memory address generation, reciprocal computation, floating point data memory upload/download, fixed point to floating point conversion and vice versa. These missing features were first implemented in the simulator. Both the assembler and the simulator are written in C language and are run from command line. This section describes them briefly.

2.7.1 Assembler

The assembler parses an assembly program of the SIMD unit and outputs instruction codes. Those instruction codes are used for verification and matrix inversion firmware together with the simulator and ModelSim. All instructions are supported along with different variants. These assembler’s output instruction codes are loaded into program memory for running the application.

During instruction verification, a minor bug was found in the instruction code generation of the special purpose register read or write command. This bug was fixed after consulting the appropriate developer of this assembler.

2.7.2 Parser

The output file of a program generated by the assembler can be used by testbench for functional verification. These testbenches are named as test vectors. Whenever there is need to run any test vector, it is done just by including that file in the testbench.

There was a need to convert this output instruction code file into purely hex file understandable by the SIMD unit. For this purpose, the author wrote a small script in python language; which is a type of parser. It gets the assembler’s output file and converts it into purely hex codes, which can be downloaded into the program memory.

2.7.3 Simulator

The simulator is used to simulate the SIMD unit for verification and matrix inversion firmware. It simulates the program code and memories by reading the hexadecimal instructions code laying in the program memory.

The following things were implemented in the RTL but not in the behavioral simulator (Inspired by [9]). A few of them, which were crucial has been updated.
Up-to-date pipeline and forwarding information

The pipeline and forwarding information in the simulator is incorrect as compared to the RTL; because the simulator does not really emulate the pipeline. It may give different number of clock cycles compared to the RTL. However, the simulator still executes the code correctly and can put the necessary stall cycles where they are needed in the code. It helps the programmer in rescheduling the program to increase the performance.

Different floating point rounding

The simulator and the RTL give slightly different results due to different rounding strategies during floating point computations. It is important to remember when comparing the simulator output with the RTL output.

All the data going or coming from the SIMD unit in the RTL are in fixed point format. The SIMD unit internally converts that fixed point data into floating point format. Whereas in the simulator, all the data being handled was already in the floating point format. The same thing is implemented in the simulator by providing fixed point data and internally converted it into floating point format.

Relative/Absolute addressing

The two addressing modes, relative and absolute are implemented in the RTL. In the beginning, the simulator was treating all bits as an absolute address (0-63). The relative addressing mode was missing in the simulator. The relative and the absolute address field is updated in the simulator the way it is intended to work in the RTL.

4-way processing

The RTL implements four parallel ways for inverting four matrices at a time. Since, there are supposed to be four antennas active. These four parallel ways are identical with exception of the reciprocal unit which is shared. There is no communication at all between these four parallel ways.

The simulator only implements one SIMD way. Ideally there should be four parallel ways in the simulator as well, but there was no need for that. Since all four parallel ways are identical and verifying one way ultimately verifies all four ways. So, only one way is tested in the RTL. At the end, it is made sure by having the same results from all the four parallel ways that there are no wiring mistakes anywhere between these four ways and other modules.

The reciprocal unit

Writing to the special register SR1 computes the reciprocal of the real part of the written values, and returns zeros for the imaginary part. After 10 clock cycles, the correct result can be read from the SR1. This thing was missing in the simulator.
and has been updated.

Automatic stalling is implemented to handle data dependencies between instructions, there is no automatic stall implemented for the special registers. The programmer is responsible for knowing when the result can be read.

**Looping**

When execution is not stalled or otherwise stopped, the following happens at every clock cycle:

```c
if(PC++ == loop_end) {
    data_mem_ptr += data_mem_ptr_step;
    if(loop_iter != 0) {
        PC = loop_start;
        loop_iter--;
    }
}
```

In any case the loop body is run once even if the loop_iter is set to 0, since the condition is only tested at the bottom of the loop. Setting the loop_iter to 1 runs the loop body twice, and so on. It gives exactly the same effect on unrolled code, except that unrolling does not update the data memory pointer after each iteration.

During verification of the instruction set of the SIMD unit, there was a bug found in the simulator for the command ADD,acc,acc,reg. This bug was fixed after consulting the designer of the SIMD unit.
Chapter 3

Verification

While designing hardware, one usually does not get the chance to market test the designed product for feedback. Improperly verified product can create a lot of troubles and may lead to failure in the market. It is very important to verify the hardware before the chip tape out. That is why; verification is the most critical and time consuming task in the whole design process.

The most important thing in verification is to apply different techniques and combinations of data for covering the corner cases. Since, this hardware is not only made for complex matrix inversion, but it can also be reused for many other operations such as complex multiplication, filtering, correlation and FFT/IFFT [4]. The author has applied different combinations of instructions for testing the SIMD unit.

3.1 Instruction Format

In the SIMD unit described in the previous chapter, there are two pipelines running in parallel, one on the left side and the other on the right side. In order to keep the both pipelines busy, the instructions set is designed in such a way that it executes two instructions at once. There are two parts of one instruction as well, one “left side” instruction and other “right side” instruction that are executed in parallel like VLIW. One instruction consists of 32 bits in toto, 18 bits for left side instruction and 14 bits for the right side instruction as shown in Fig: 3.1. Left and right side instructions are separated by a separator ||.

3.1.1 Instruction encoding

The general format of the instruction encoding is a bit difficult to explain because of different number of source operands. Some instructions only have one source operand and one destination operand; some have two source operands and one destination operand where they store the result after doing some operation on the source operands. We will discuss by taking a format which is most relevant in
Figure 3.1. SIMD instruction format.

almost all the instructions.

First three left most bits of the left side instructions are the op code for any command. These three bits are common in all the left side instructions. The next two adjacent bits (from left to right) indicate the destination accumulator register. These two bits are also common in all those instructions which have accumulator register as a destination operand. Next five adjacent bits indicate the source operand in instructions which have two source operands. Next three adjacent bits are used for changing the sign of the imaginary part or the real part or negating both the real and the imaginary part in the source operand. Right most five bits of the left side instruction indicate the source operand irrespective of this whether the instruction has two source operand or one source operand. Table: 3.1 shows a left side instruction encoding in general.

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op code</td>
<td>[31:29]</td>
<td>Specify the op code for left side instructions.</td>
</tr>
<tr>
<td>Source operand1</td>
<td>[26:22]</td>
<td>Specify the second source operand in instructions which have two source operands.</td>
</tr>
<tr>
<td>Operand2’s signs</td>
<td>[20:19]</td>
<td>2’b01: Change sign of imaginary part.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2’b10: Change sign of real part.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2’b11: Negate signs of both real and imaginary part.</td>
</tr>
<tr>
<td>Source operand2</td>
<td>[18:14]</td>
<td>Specify the source operand for all instructions.</td>
</tr>
</tbody>
</table>

Table 3.1. Left side instruction encoding.

Similarly, in the right side instruction the most significant two bits indicate the op code of instruction. These two bits are common in all the right side instructions. The next five adjacent bits from left to right are for the destination
operand, which are also common for all the commands on the right side. Next bit is used for indicating absolute or relative addressing in memory commands and it will be don’t care for rest of commands. Least significant five bits indicate the source operand. Table: 3.2 shows a right side instruction encoding in general.

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address mode</td>
<td>[6]</td>
<td>1'b0: Relative address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1'b1: Absolute address</td>
</tr>
<tr>
<td>Source operand</td>
<td>[5:0]</td>
<td>Specify the source operand.</td>
</tr>
</tbody>
</table>

Table 3.2. Right side instruction encoding.

As mentioned earlier, the above two formats are the most relevant formats for all the instructions, because the positions of op code, source operand and destination operand is same for all instructions except some differences in the positions of bits for changing the signs of source operands. A complete list of instructions encoding and their operations can be found in Appendix A.

### 3.2 Pipeline Delays

Different instructions in the SIMD unit takes different number of clock cycles to complete. In a program one instruction may have data dependency on the previous instruction for using some value updated by the previous instruction. Current instruction has to wait until previous instruction finishes, if there is data dependency. Although the control unit has the ability to automatically stall parts of the pipeline and inserts as many cycles as necessary until it is safe to run the next instruction. But it is important to know the pipeline delays for code optimization.

A stall cycle is equivalent to manually inserting as many NOPs as necessary in the program between the two instructions, which of course increases the execution time. By knowing the data dependency delays one can arrange the code in such a way that can avoid stalls and thereby avoid the increased execution time. Ideally the execution time should be equal to the number of instructions, but often this is impossible to achieve in practice. Table: 3.3 shows the data dependency delays for going from one instruction to an other. (Inspired by [10]).

#### 3.2.1 Stalling or manual NOPs

For getting more sense of automatic stalling and manual NOPs and their advantages in terms of memory usage or timings, we see it by examples. The following two sequences are exactly equivalent in terms of execution time (number of clock
cycles). The only difference between sequence 1 and sequence 2 is that sequence 1 occupies less program memory.

--- Example 3.1: Stalling ---

\[
\begin{align*}
\text{mul } a0, r10, r10 &; \text{ sequence 1} \\
\text{mul } a0, a0, r10 &\\
\text{add } a1, r11, r11 &\\
\text{add } a1, a1, r11 &
\end{align*}
\]

\[
\begin{align*}
\text{stall cycle} &\\
\text{mul } a0, a0, r10 &\\
\text{add } a1, r11, r11 &\\
\text{add } a1, a1, r11 &
\end{align*}
\]

The following code sequence performs the same work, but the programmer has been aware of the pipeline delays and rearranged the instructions to avoid stalls. Sequence 3 executes in four clock cycles rather than six. (Inspired by [10]).

--- Example 3.2: Manual NOPs ---

\[
\begin{align*}
\text{mul } a0, r10, r10 &; \text{ sequence 3} \\
\text{add } a1, r11, r11 &\\
\text{mul } a0, a0, r10 &\\
\text{add } a1, a1, r11 &
\end{align*}
\]
If “|| right instruction” is omitted, it will default to “|| NOP”. But for no operation on the left side, one has to give NOP command explicitly. For example in the following commands first instruction automatically put NOP on right side whereas for NOP on the left side, one has to express explicitly.

---

**Example 3.3: NOP commands**

1. stl 10, r8
2. nop || move r19, a3

---

### 3.3 Instruction Verification

This SIMD unit is supposed to work with the Senior processor (refer to section: 5.1). In order to keep the verification simple and easy to handle, a Senior model is developed in a RTL testbench. This Senior model behaves exactly the way the Senior processor is supposed to behave. It uploads the data and program into the SIMD data and program memory respectively. It also configures the SIMD unit by IO_WRITES and tells the SIMD unit to run the program. When the SIMD is finished with its computation, it lets the Senior know by sending an interrupt. Finally, the Senior model asks the SIMD unit to send back the results from the data memory.

All the data going or coming from the SIMD unit is in 16 bits fixed point format. The SIMD unit internally converts that fixed point into 20 bits floating point with a sign bit, 6 exponent bits and 13 mantissa bits and then stores it into the data memory. It has the capability to saturate the overflowed data to maximum and minimum values which are 0x7FFF and 0x8000 respectively.

A complete list of left side instructions and right side instructions of the SIMD unit is given in Appendix A. These instructions have been categorized into five types of programs which almost covers all possible combinations and corner cases. Those five categories are addition, multiplication, multiply-accumulate, reciprocal and memory load/store. The author proceed by taking one by one and then discuss how each instruction was verified and the problems faced during the verification.

#### 3.3.1 Addition operation

There are two instructions available for addition operation. In first instruction, both source operands can be registers and in the second instruction one operand can be register and the other can be accumulator. Both of these instructions provide the facility of changing the signs of the operand’s imaginary part, real part or both. Fig: 3.2 shows the datapath of the SIMD unit for addition operation [11].
Addition operation has been verified by making different combinations of instructions. A few of them are simple addition of source operands or negating first the imaginary or the real part of one operand. In one combination, both the real and the imaginary part are negated and kept the second operand as it is or negated the real or the imaginary part of the second part as well. All the possible combinations as per instruction validity have been verified. All of the above operations have also been verified by having one operand as an accumulator.

While running test vectors for addition, a bug found in the RTL for command ADD acc,~acc,~reg. The real part of the source accumulator was not negating. This bug was fixed in the control unit.
3.3 Instruction Verification

3.3.2 Multiply operation

Like addition, there are also two instructions available for multiply operation. In first instruction both source operands are registers and in the second instruction one operand can be a register and the other can be an accumulator. Both of these instructions provide the facility of changing the signs of the operand’s imaginary part, real part or both exactly like add instructions.

Complex numbers do multiplications in two steps in backend. First it uses multiplier and then passes those outputs to an adder. After addition of multiplier’s output we get the final result. Fig: 3.3 shows the datapath of the SIMD unit for multiply operation [12].

Figure 3.3. Multiply datapath.
Multiply operation has also been verified by making different combinations of instructions. A few of them are simple multiplication of source operands or negating first the imaginary part or the real part of one operand. In one combination, both the real and the imaginary parts are negated and kept the second operand as it is or negated the real or the imaginary part of the second part as well. All the possible combinations as per instruction validity have been verified. All of the above operations have also been verified by having one operand as accumulator.

During verification of the multiply operation, there was a deficiency found in the stall logic which can be improved. If the last instruction of the program which is at the address pointing by the program_end register stalls, it will be lost and never executed. This implies to all those instructions at the end of program which need stalls. For example, the below program consists of five instructions. The STG is the last instruction and at the same time it must stall because the data is not available. The STG instruction is simply lost. The same applies to other instructions at the last position in the program if it has to stall.

Example 3.4: Stalling last instruction

```
1. nop || ldg r31,0 ; absolute address 0
2. nop || ldg r30,1 ; absolute address 1
3. mul a1,r31,r30
4. nop || move r0,a1
5. stg 5,r0
```

In order to bypass this hole until it is fixed, one must make sure that the last instruction in the program does not stall. The easiest way is simply to append a NOP at the end, because NOP never stalls and even if it did, it would not matter that, it would be lost since a NOP does no useful work anyway.

Example 3.5: Avoiding stall in last instruction

```
1. nop || ldg r31,0 ; absolute address 0
2. nop || ldg r30,1 ; absolute address 1
3. mul a1,r31,r30
4. nop || move r0,a1
5. stg 5,r0
6. nop
```

### 3.3.3 Multiply-accumulate operation

There is only one instruction for multiply-accumulate operation, which is MAC. The MAC instruction uses general purpose registers as source operands and ac-
cumulator as destination operand. It gives the facility of changing the signs of the real and the imaginary parts of operands. Fig: 3.4 shows the datapath for multiply-accumulate operation [13].

Like addition and multiplication, the MAC operation has also been verified by making all the possible combinations of operand’s signs. One bug was found while performing the MAC operation; the sign bits in the final adder were undefined. This bug was fixed by defining their signs.

Figure 3.4. Multiply-accumulate datapath.
3.3.4 Reciprocal operation

The SIMD unit takes reciprocal of only the real part of a complex number and returns zeros for its imaginary part, because multiplying complex number with its conjugate gives the scalar value for the denominator. The reciprocal takes place by first writing the value into special register and then reading it back after 10 clock cycles. This is the most time consuming operation of this SIMD unit.

The reciprocal operation has been verified by number of ways. For example inverting simple data, inverting data by first changing the imaginary or the real sign or both.

Test vectors for the reciprocal operation also cover a number of other instructions which are move from special registers to general purpose registers and vice versa, accumulator move to special registers and memory load/store instructions.

The reciprocal of zero in the RTL is different from the SIMD simulator. Because in the simulator, zero is represented as +0 even after negating, it remains as +0; whereas in the RTL, zero is represented as +0, but after negating, it becomes -0. If +0 is inverted in the RTL, the value nearest to +infinity that can be represented is saturated to 0x7FFF in fixed point. Similarly, if -0 is inverted, the value nearest to -infinity that can be represented is saturated to 0x8000 in fixed point.

3.3.5 Memory load/store

Memory can be accessed locally or globally. The SIMD unit provides separate instructions for both the local/relative and the global/absolute memory access. The relative address generates by adding data memory pointer with offset, whereas for the absolute address, the input offset is directly treated as an absolute address.

The address field in the load and store instructions is 7 bits. In the RTL, the highest bit indicates whether it is a relative (0) or absolute (1) address. In the assembler language, this is indicated with an L or G suffix on the LD and ST instructions.

For a relative address, the remaining 6 bits are interpreted as a signed offset from the data memory pointer, data_mem_ptr-32 to data_mem_ptr+31. While in the absolute address, the remaining 6 bits are interpreted directly as an absolute address (0-63).

The data memory pointer is changed each time PC reaches the loop_end address as indicated by the step size. A load or store at loop_end will use the old data memory pointer while the following instruction at loop_start or loop_end+1 will use the new data memory pointer.

The relative and the absolute memory access instructions are tested by writing data at different locations and then read back. It was working perfectly without
any problem.

3.4 Code Coverage

At the end of verification process, one must know how much one’s testbenches are thoroughly testing the design. Many tools provide code coverage feature for ensuring the quality and thoroughness of tests. This feature of ModelSim has also been used for getting the code coverage statistics. According to ModelSim’s code coverage report, testvectors are covering overall 99.6% portion of this SIMD unit, in which 94.9% is core coverage. It covered all branches, statements, conditions and toggle states.
Chapter 4

Matrix Inversion Firmware

To overcome the limitation of resources, multi user communication is used. Multiple streams of inputs and outputs over a single communication channel having multiple sub channels are commonly referred to as MIMO. Different multiplexing techniques help in carrying multi user data through a single channel. By using this idea, higher data throughput can be achieved without additional bandwidth or transmit power. It is made possible by higher efficiency i.e. more bits per second and diversity [14].

There have been various MIMO signaling schemes such as space-time coding, spatial multiplexing and beamforming exploiting different freedoms of multi-antenna system. Most of these algorithms require complex matrix inversion. Among the most extensive tasks, complex matrix inversion is the one which is most complex and MIPS demanding [4].

![MIMO communications](image)

**Figure 4.1.** MIMO communications.

4.1 Classical Matrix Inversion

Traditionally matrix inversion for larger matrices is implemented by QR factorization. By QR factorization, upper triangular matrix R is generated from the original matrix and then result is computed by back substitution [4]. There are many ways to compute QR decomposition for example Householder QR, Givens
QR methods and Gram-Schmidt transform [15].

Recently, Squared Givens Rotations (SGR) got attention for QR decomposition for hardware implementation. It decreases the number of multiplications to half and eliminates the square-root operation. At the same time, parallelism is still there in SGR which helps in mapping to parallel processing hardware for getting higher performance [4].

Systolic array is a traditional architecture to implement QR decomposition for achieving higher performance. However, it consumes huge silicon area and does not scale very well on increase of matrix size. Another architecture, Linear array is more scalable than the traditional systolic array. There is another solution presented in [16], in which similar systolic array. There is another solution presented in [16], in which similar systolic array. There is another solution presented in [16], in which similar systolic array. There is another solution presented in [16], in which similar systolic array. There is another solution presented in [16], in which similar systolic array. There is another solution presented in [16], in which similar systolic array. There is another solution presented in [16], in which similar systolic array. There is another solution presented in [16], in which similar systolic array. There is another solution presented in [16], in which similar systolic array. There is another solution presented in [16], in which similar systolic array. There is another solution presented in [16], in which similar systolic array. There is another solution presented in [16], in which similar systolic array. There is another solution presented in [16], in which similar systolic array. There is another solution presented in [16], in which similar systolic array. There is another solution presented in [16], in which similar systolic array. There is another solution presented in [16], in which similar systolic array. There is another solution presented in [16], in which similar systolic array. There is another solution presented in [16], in which similar systolic array. There is another solution presented in [16], in which similar systolic array. There is another solution presented in [16], in which similar systolic array. There is another solution presented in [16], in which similar systolic array.

4.2 Matrix Inversion Algorithm

Systolic array-based QR decomposition is good for large matrix inversion. In our SIMD unit, baseband signal processing is involved which contains small matrices. Because of this, the QR decomposition is not that much efficient for this design [4].

The matrix inversion algorithm for the SIMD unit is a modified version of direct analytical approach. Simple direct analytical approach was not good for $4 \times 4$ matrix inversion due to stability issues which are explained in next topic.

4.2.1 Direct analytic matrix inversion

Direct analytical approach is a simple way to compute matrix inversion. $H^{-1}$ is computed by multiplying the adjugate matrix $(C_{ij})^T$ with the inverse of determinant $|H|$ of the original matrix.

$$
H^{-1} = \frac{1}{|H|} (C_{ij})^T = \frac{1}{|H|} \begin{pmatrix}
C_{11} & C_{12} & \cdots & C_{1j} \\
C_{21} & \ddots & & C_{2j} \\
\vdots & & \ddots & \vdots \\
C_{i1} & \cdots & \cdots & C_{ij}
\end{pmatrix}
$$

The inverse of $2 \times 2$ matrix would be like:

$$
H^{-1} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}
$$
It can be seen by a real example:

---

**Example 4.1: 2×2 Matrix inversion**

\[
H = \begin{bmatrix}
4 & 6 \\
7 & 0
\end{bmatrix}
\]

\[
H^{-1} = \frac{1}{-42} \begin{bmatrix}
0 & -6 \\
-7 & 4
\end{bmatrix} = \begin{bmatrix}
0 & 0.14286 \\
0.16667 & -0.09524
\end{bmatrix}
\]

---

Although the direct analytical approach is simple to implement but it increases complexity very much with increase in matrix size, which makes it difficult to scale. This design is focused on 4×4 matrix due to 4 antennas (four sending antennas and four receiving antennas). Thus, matrices of maximum size 4×4 or below are considered. In this case scalability is not that much important due small matrix size. For smaller matrices, the numbers of arithmetic operations of analytical approach are significantly smaller than the QR decomposition. That is why, the analytical approach is efficient for computing inverse of smaller matrices and it is also easy to be mapped to programmable hardwares [4].

However, it is found that for 4×4 matrix inversion, the direct analytical approach is not very stable due to a lot of subtractions which may cause cancellations. The direct analytical matrix inversion is sensitive to finite-length errors. This can significantly affect the performance if there are not enough bits for numerical representation. In order to avoid this drawback, a slightly different approach is used called Blockwise Analytic Matrix Inversion [4].

### 4.2.2 Blockwise analytic matrix inversion

In blockwise analytical approach, 4×4 matrix is subdivided into four 2×2 matrices, and then based on these 2×2 matrices the inverse of 4×4 matrix is computed. These 2×2 inversions are computed using the direct analytical approach. For example, to compute the inverse of 4×4 matrix H, it will be first subdivided into four 2×2 matrices A, B, C and D.

\[
H = \begin{bmatrix}
a & b & c & d \\
e & f & g & h \\
i & j & k & l \\
m & n & o & p
\end{bmatrix}
\]

\[
A = \begin{bmatrix}
a & b \\
e & f
\end{bmatrix}, \quad B = \begin{bmatrix}
c & d \\
g & h
\end{bmatrix}, \quad C = \begin{bmatrix}
i & j \\
m & n
\end{bmatrix}, \quad D = \begin{bmatrix}
k & l \\
o & p
\end{bmatrix}
\]
The inverse of $H$ can be computed as:

$$H^{-1} = \begin{bmatrix} A^{-1} + A^{-1}B(D - CA^{-1}B)^{-1}CA^{-1} & -A^{-1}B(D - CA^{-1}B)^{-1} \\ -(D - CA^{-1}B)^{-1}CA^{-1} & (D - CA^{-1}B)^{-1} \end{bmatrix}$$

By finding the inverse of $2 \times 2$ matrix and some additional computation, the inverse of $4 \times 4$ matrix can easily be computed. Each $2 \times 2$ matrix inversion involved one $1/x$ computation for computing determinant. This $1/x$ computation is the most time taking operation of this SIMD unit.

The blockwise analytical approach is more stable than the direct analytical approach due to less number of subtractions involved. In this method, instead of inverting $4 \times 4$ matrix, only $2 \times 2$ matrices are inverted. So, there are less numbers of subtractions involved and less risks of cancellation. This also requires less number of bits for precision [4].

### 4.3 4×4 Matrix Inversion Implementation

The blockwise analytical approach for $4 \times 4$ matrix is first simulated and tested with Matlab. This Matlab code can be seen in Appendix B. The Matlab code is consist of three functions, but the firmware is just one big function, since, there is no support for subroutines. In first step, all code pasted together in the same way and roughly same order as the firmware was supposed to do it, to get a feeling for what needs to be done and how.

At the end, the registers are simulated in Matlab by naming Matlab variables as r1, r2, a0, a1 etc. Memory load/store are also simulated to verify that the algorithm is correct before writing the assembler code. After doing all this, it becomes very easy to translate the Matlab code into assembler language, since it is known exactly which registers should be used and when to load/store the data. (Inspired by [10]).

#### 4.3.1 Memory utilization and layout

Each element of the matrix is loaded and stored individually in the data memory which means it does not matter how the data is organized. The memory layout only affects the offset in the load and store instructions. The most straightforward way used to store a matrix is to store each matrix row by row as 16 consecutive values in the memory. That is, LDL r0,0 will load the upper left element of the $4 \times 4$ matrix, LDL r0,3 will load the upper right element, LDL r0,12 will load the lower left element and LDL r0,15 will load the lower right element.

It is not possible to keep the entire $4 \times 4$ matrix and all intermediate values in registers all at the same time. Only parts of the matrix are kept in registers. Since, the data cannot be overwritten in memory that is needed later. This thing is resolved by putting the first matrix at address 16-31, the next one at address
32-47 and saved the result of first one to address 0-15, and so on. In this way, the current matrix is not overwritten as shown in Fig: 4.2. (Inspired by [10]).

![Figure 4.2. Memory utilization.](image)

Data upload to the memory will fill the memory from address 16 and upward, for example, to address 47 if there are two matrices. Result download from the memory will be downloaded from address 0 up to address 31. The data memory pointer will be updated automatically in loop so that the SIMD unit can read a new matrix on every iteration. The data memory pointer step register should be set to 16.

![Figure 4.3. Memory layout.](image)

One complex number consists of 16 bits real part and 16 bits imaginary part.
The SIMD unit accepts 16 bits fixed point data and converts it into 20 bits floating point data before it is written into memory. This means, one complex number takes 40 bits (20 bits for the real part and 20 bits for the imaginary part) for saving into the memory. Since, this is a 4-way SIMD, there are 4 matrices inverting at a time. This will definitely demand four times wider memory and registers. Ultimately, the total data memory size is $160 \times 256$. This can be visualized from Fig: 4.3.

It is possible to invert plenty of matrices by first uploading all, and then start the SIMD unit. It will process all matrices by itself in a loop and only notify to Senior (a control processor) once it is finished with all of them. The need for inverting plenty of matrices is that, matrix inversion is one way used by the receiver to compute an estimate of how the radio channel distorts the received data and how the data is received by the different antennas. The receiver uses training data for computation which in some cases is interleaved with the real data like in LTE system. In an OFDM system, the subcarriers are in a sense independent, and the assumption is that there are hundreds of subcarriers. (Inspired by [10]).

During inversion of multiple matrices, a bug appeared related to loop_end and prog_end. If the loop_end register is set to some arbitrary address which is less than the prog_end and loop_iter is set to zero, then it updates the data memory pointer when PC reaches to the loop_end address; which it should not do. For example, if there is a program of 0x0014 lines and the loop_end is set to 0x000C but the loop_iter is zero. It should not update the data memory pointer. This bug is fixed.

4.3.2 Code optimization

The firmware was first started with much unoptimized version. It was consisted of 203 instructions and was taking 304 clock cycles to invert one $4 \times 4$ matrix. This result was not acceptable. Since, the main focus was to achieve the higher performance.

In order to make the firmware efficient, different instructions were scheduled/rearranged in such a way, that it avoids stalls and NOPs. For example, the $1/x$ computation takes 10 clock cycles to compute, it is possible to overlap this with other computations and memory operations. Following example shows the code overlap.

---

**Example 4.2: Overlapped code fragment**

```assembly
move sr1,a0 || ldl r5,25 ; load sr1, load b
nop || ldl r6,28 ; load c
nop || ldl r7,29 ; load d

; Load $2 \times 2$ Matrix B
```
In the above example, at first the special register is loaded for taking the inverse of operand. Now instead of waiting 10 clock cycles with NOPs, memory load or some other moves have been overlapped. In this way efficiency is increased.

Beside this, there were some other tactics used to perform as much computation as possible with fewer number of instructions. For example, it is tried to use ADD/MUL instructions with one A-register operand to reduce the number of moves from A- to R-registers, and kept the both left and right pipelines busy. Whenever required, used all available registers to avoid overwrite values that are still in use.

Wherever it was possible to eliminate NOPs, this was done by moving instructions up or down. Let us proceed by taking different fragments of code and see how they were optimized.

There were first five instructions in the code:

---

**Example 4.3: Code optimization**

```
; Unoptimized code
nop  ||  ldl r0,16    ; load a
nop  ||  ldl r1,17    ; load b
nop  ||  ldl r2,20    ; load c
nop  ||  ldl r3,21    ; load d
mul a1,r1,r2          ; b·c

; Optimized code
nop  ||  ldl r1,17    ; load b
nop  ||  ldl r2,20    ; load c
da  ||  ldl r0,16    ; load a
mul a1,r1,r2 ||  ldl r3,21 ; b·c, load d
```
In the above fragment of code, the "ldl r1" and "ldl r2" instructions are critical because their results are used immediately by the "mul" instruction. The "ldl r0" and "ldl r3" are less critical. By moving the critical instructions up, the "mul" instruction can be executed earlier. There is one instruction between to allow for pipeline delay ldl->mul. Also, the "mul" and the final "ldl" can be executed in parallel since there is no dependency between them (See section: 3.2 for pipeline delays). (Inspired by [10])

Here is another fragment of code:

--- Example 4.4: Unoptimized code ---

```plaintext
nop || move r30, sr1 ; get 1/(a·d-b·c)
mul a0,r30,r29
nop || move r30,a0

; W = Multiply 1/(a·d-b·c) and adjoint of A
; W in r12, r13, r14, r15
mul a0,r30,r3
nop || move r12,a0
mul a0,r30,-r1
nop || move r13,a0
mul a0,r30,-r2
nop || move r14,a0
mul a0,r30,r0
nop || move r15,a0

; X = C·W in r0, r1, r2, r3
mul a0,r4,r12
nop || move r0,a0
mul a1,r5,r14
add a1,a1,r0
nop || move r0,a1
mul a0,r4,r13
nop || move r1,a0
mul a1,r5,r15
add a1,a1,r1
nop || move r1,a1
```

In the above code, the both pipelines are not busy at a time and it also consists of many instructions. See below, how the number of instructions have been reduced by putting workload on the both pipelines:
Example 4.5: Optimized code

```assembly
nop || move r30, sr1 ; get 1/(a·d-b·c)
mul a0,r30,r29

; W = Multiply 1/(a·d-b·c) and adjoint of A
; W in r12, r13, r14, r15
nop
mul a1,a0,r3
mul a2,a0,-r1
mul a3,a0,-r2 || move r12,a1
mul a1,a0,r0 || move r13,a2

; X = C·W in r0, r1, r2, r3
mul a0,r4,r12 || move r14,a3
mac a0,r5,r14 || move r15,a1
mul a1,r4,r13
mac a1,r5,r15
...
```

In the above optimized code, the both pipelines are kept busy as much as possible. When optimization is done, the code may be virtually impossible for anyone to understand, including the programmer himself; but that is an unfortunate consequence of code optimizations. After optimization, the firmware consists of 118 instructions and it takes 152 clock cycles.
Chapter 5

Senior Integration and Synthesis

This SIMD unit is specifically designed for performing the efficient matrix inversion. Although it can be used as a main processor for other purposes, but in this case it will work as a coprocessor with some control processor. For this purpose, a Senior as a control processor is integrated with the SIMD unit to work as a master device. The SIMD unit and the Senior communicate through the SIMD interface.

5.1 The Senior Processor

The Senior processor is a DSP processor. It executes one task at a time. The native width of all registers and addresses is 16 bits. Our focus is to have understanding of only those modules which are required for integration.

The most simplified way to connect hardware with the Senior core is through general I/Os using in and out instructions. It is possible to connect up to 64 peripherals; few of them are tightly coupled as shown in Fig: 5.1. For more information about the Senior, refer to the Senior documentation [17].

5.2 Senior Integration

In order to keep simplicity in the beginning, there was a simple testbench having the SIMD interface, which was configuring, uploading data/program into the SIMD unit and after completion of computation, reading back results from the SIMD data memory. The testbench is replaced entirely with the Senior and a program running on the Senior. The Senior program performs the ‘io_writes’ that the testbench was performing. Somewhere in the Senior’s memory, there is an input data and the SIMD program. The Senior program uploads the data and the program into the SIMD unit, then start the SIMD unit by writing a suitable value into its main control register (IOR_CONTROL). Then the Senior program sees if
it has finished by an interrupt from the SIMD unit. When the SIMD finishes, the Senior program reads out data from the SIMD memory into Senior memory. Fig: 5.2 shows the Senior and the SIMD integration as off-chip devices.

The SIMD unit is intended as an on-chip peripheral, but for testing purposes, it is used as an off-chip peripheral to the Senior chip. The off-chip interface basically works like the on-chip interface. The internal 6 bits address bus of the Senior is connected to the external 6 bits address bus through a couple of flip-flops. A simple out instruction always activate the external I/O interface. So, writing to the off-chip peripheral interface can be performed by an out instruction with an address between 0 and 63. The SIMD unit use the incoming 6 bits address to select which register should be forwarded to its output port (to the Senior data_i).

Reading from the off-chip peripheral interface can be done by using an in instruction through port 16. There is no strobe of any kind will be activated when using an in instruction. The Senior read/write to the SIMD unit has been done by two ways, first by simple I/O port and second by DMA.
5.2 Senior Integration

Figure 5.2. Senior and SIMD integration.

5.2.1 Read/write through I/O ports

In order to keep the testing simple, the SIMD unit was first tested simply by the Senior I/O ports. The Senior processor configures the SIMD unit, uploads input data and program into the SIMD unit and starts the execution exactly like it was doing by testbench. After this, it waits for an interrupt from the SIMD unit indicating the completion of task.

The interrupt service routine is located at address 0x108, considering the base address of interrupt vector table as 0x100. In interrupt service routine, the Senior reads the data coming from the SIMD unit through port 0x10. When it finishes reading, it writes 0 to port 0x18 to stop the simulation.

5.2.2 Read/write through DMA

Read/write through the I/O port takes much time. In order to reduce this time, the DMA was chosen to perform this task. The Senior data send and receive is done using the DMA channel 6. It is capable of sending data to the peripheral interface through data_o[16:0] (data_o[16] is the strobe), or receiving data through data_i[16:0] (data_i[16] is the strobe). Though, it is also necessary to give the DMA engine ownership of the peripheral interface.

Before write to the SIMD unit, it is necessary to configure the outer I/O mux. For example, 0x8013 write to port 0x47 means “let the DMA access the interface” (bit 15 set) and it should use port 0x13 (bits 5-0 is 0x13).

The DMA is also required to configure before data send/receive. There is a task packet used to setup the DMA transfer channel. The task packet is of 16 bits because the DSP core has a general I/O of 16 bits. It configures the source and destination ports. For more information about the DMA, refer to the “DMA User Manual” [18].
When the SIMD unit finishes computation, it interrupts the Senior. The interrupt service routine just clears the interrupt flags. The DMA reads the results from the SIMD memory after exit from the interrupt service routine.

### 5.3 RS232 Interface

It was not possible to test the Senior and the SIMD unit standalone on FPGA. There was a need of some interface for communication with the computer for program downloading and reading back the results. For this purpose, an off-the-shelf RTL is used in which the Senior communicates with the computer through a MCU (Micro Controller Unit), SPI (Serial Peripheral Interface) and Wishbone interface. The MCU is the same one that is included in the PyXDL distribution available at [19].

The Senior contains SPI interface, which is connected with the SPI master. This SPI master is connected with RS232 port through a wishbone and a microcontroller unit which controls the traffic on different paths. This can be visualized in Fig: 5.3.

There is another external memory connected with the Senior, which is used for saving the results. One can read back the results from this external memory. For testing purposes, the Senior writes the resulted data which has been read from the SIMD unit into external data memory. An already implemented application is used for data/program downloading into the Senior and then reading back the results from the external memory through RS232 interface.

![Figure 5.3. RS232 interface with the system.](image)

After integrating all these units, the final simulation was needed to run. Before running the final simulation, the author updated the test bench with wishbone read and write tasks. In the final behavioural simulation, it was verified that the system is bug free.
5.4 Synthesis

Synthesis is an automated design process that converts a high level description of design into an optimized hardware. Logic synthesis uses a standard cell library which have simple cells, such as basic logic gates like and, or, and nor, or macro cells, such as adder, multiplexers, memory, and flip-flops. Standard cells put together are called technology library [20].

The SIMD unit circuit description is written in Verilog HDL (Hardware Description Language). Before synthesis, the architectural description of design was fully understood and verified functionally. The synthesis process checks code syntax and analyzes the hierarchy of the design. The resulting netlist(s) is saved as an NGC (Native Generic Circuit) file.

The RTL is synthesized at 25 MHz with speed grade 10 on Virtex-4 FPGA mounted on Avnet LX evaluation board by using Xilinx tools [22]. The device number is xc4vlx60 containing package number ff668. It can be viewed in Fig: 5.4.

![Virtex-4 evaluation board](image)

**Figure 5.4.** Virtex-4 evaluation board.

In the beginning, the floating point simulation models were generated by Xilinx Coregen for floating point multiplication, subtraction, division and floating point to fixed point conversion and vice versa. At the time of synthesis, the synthesis tool used the NGC files which contain the real netlists for the floating point modules. The actual contents of the modules are replaced by the NGC files.

While generating the NGC files, the pipeline stages for the adder and the multiplier are selected as 0. Because, it would be used in an ASIC, and it is very difficult to schedule assembler code if one has to wait 3 clock cycles between MUL->MAC etc. The other units can in principle be pipelined arbitrarily. There are 3 stages
selected for the divider which gives 10 clock cycles latency for sr1 and if the num-
ber of pipeline stages changes, the assembler code must have to be changed. The
3 stages for the inverter is as a trade-off between logic depth and latency. There
is more work inside an inverter than in an adder or multiplier, that is why; it was
not sufficient to use 0 pipeline stages.

It is not good to use too many pipeline stages because there are already 10 cy-
cles for sr1, increasing 3 to 4 will increase sr1 to 11. On the other hand, decreasing
to 2 pipeline stages will only decrease sr1 to 9 cycles which is almost the same as
10. So, 3 are just fine.

The pipeline length for the converters is set to zaro, but it is not important
since it only affects data upload and download. (Inspired by [10]).

Implementation process consists of three steps:

- Translate
- Mapping
- Place and route

**Translate**
This process combines all the input netlists (NGC) and creates the NGD (Native
Generic Database) file. This file describes the design in terms of logical elements
like Logic Gates, Decoders, Flip-Flops and RAMs etc. Here, defining constraints
is nothing but, assigning the ports in the design to the physical elements (ex. pins,
switches, buttons etc) of the target device and specifying time requirements of the
design. This information is stored in a file named UCF (User Constraints File) [21].

**Mapping**
In this step, the design is mapped to different sub blocks such that they can be
fit into the FPGA logic blocks. It fits the logic defined by the NGD file into
the targeted FPGA elements (Combinational Logic Blocks (CLB), Input Output
Blocks (IOB)) and generates an NCD (Native Circuit Description) file. This file
physically represents the design mapped to the components of FPGA [21].

**Place and route**
The place and route process places the sub blocks from the map process into logic
blocks according to the constraints and connects the logic blocks. For example, if
a sub block is placed in a logic block which is very near to I/O pin, then it may
save the time but it may affect some other constraint. So, trade-off between all
the constraints is taken into account by the place and route process. The PAR
step takes the mapped NCD file as input and produces a completely routed NCD
file as output which can be analyzed for timing [21].

After completion of above steps, the design can be loaded on the FPGA. Be-
fore downloading, it must be converted to a format, that the FPGA can accept
it. BITGEN program deals with the conversion. The routed NCD file is then given to the BITGEN program to generate a bit stream (a .BIT file) which is used to configure the target FPGA device. This was done by an iMPACT application through USB (Universal Serial Bus) interface.

### 5.5 Area Utilization

Sometimes it might be of one’s interest to know, how much area are occupied by one’s RTL. The area utilization of the SIMD unit and the whole system is analyzed below separately.

<table>
<thead>
<tr>
<th>Module</th>
<th>Slices</th>
<th>Slice Reg</th>
<th>LUTs</th>
<th>LUTRAM</th>
<th>BRAM</th>
<th>DSP48</th>
</tr>
</thead>
<tbody>
<tr>
<td>senior/acc</td>
<td>54</td>
<td>65</td>
<td>58</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>senior/dma</td>
<td>425</td>
<td>363</td>
<td>463</td>
<td>17</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>senior/dsp_core</td>
<td>3895</td>
<td>1898</td>
<td>5939</td>
<td>20</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>senior/hif</td>
<td>74</td>
<td>88</td>
<td>108</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>senior/ic</td>
<td>32</td>
<td>20</td>
<td>47</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>senior/mem</td>
<td>363</td>
<td>111</td>
<td>616</td>
<td>0</td>
<td>80</td>
<td>0</td>
</tr>
<tr>
<td>senior/serio</td>
<td>46</td>
<td>77</td>
<td>44</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>senior/per_rf</td>
<td>84</td>
<td>64</td>
<td>117</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>senior</td>
<td>5409</td>
<td>3013</td>
<td>7840</td>
<td>43</td>
<td>81</td>
<td>3</td>
</tr>
<tr>
<td>simd/core</td>
<td>10952</td>
<td>3087</td>
<td>18530</td>
<td>1950</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>simd/dm</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>simd/pm</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>simd</td>
<td>11273</td>
<td>3087</td>
<td>18852</td>
<td>1950</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>dmem</td>
<td>15</td>
<td>12</td>
<td>16</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>spin</td>
<td>46</td>
<td>62</td>
<td>81</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>wb_mcu</td>
<td>326</td>
<td>264</td>
<td>449</td>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>19713</td>
<td>6438</td>
<td>28027</td>
<td>2001</td>
<td>90</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5.1. Area utilization.

After synthesis, the .bit file was downloaded into the FPGA by an iMPACT software. The Senior program, which contains both the program and data of the SIMD unit, is downloaded into the Senior’s program and data memory through serial interface. The application first resets the entire hardware and then downloads the data and program into the Senior’s memories after clearing it, starts the Senior’s core, waits for some seconds and then reads back the results. Unfortunately, at first, the system was not working. The author used different techniques for debugging the system, which are described in the next section.
5.6 Debugging Techniques

There were many things in the system which can be suspected, like downloading application running on computer, RS232 interface, MCU, wishbone interface, SPI interface, Senior, SIMD unit or external memory. All of these things were verified step by step.

5.6.1 Reading Senior memory

The first thing was to check whether program is downloaded properly into the Senior’s memory or not. By ensuring this, one can assume that the downloading application, RS232 interface, MCU, wishbone interface and SPI interface are working properly; because the program download passes through all these interfaces.

In order to do this, the author added a function for program memory read back into the downloading application. This function was added right after the program downloads function. When this application was run, it confirmed that the program is downloaded perfectly and all of the above mentioned interfaces are bug free. Now the only things left which can be culprit are the Senior, the SIMD unit or the external memory.

For saving time and quickly debugging the system, it was assumed at first that the Senior and the SIMD unit are bugs free and problem is in the external memory. In order to ensure this, a very simple program was written, which only writes few values into the external memory and then read back those values by downloading application running on computer. This simple program confirmed that there is no bug in writing or reading back from the external memory. This implied that the bug is somewhere in the Senior or in the SIMD unit.

5.6.2 NGC simulation

The NGC simulation was performed to verify that the functionality is correct after synthesis. It uses a post-synthesis simulation model (a structural UNISIM-based VHDL or Verilog file) that is generated by NetGen [23]. This process is used for debugging synthesis-related issues.

In the NGC simulation, the system was still not working. After examining the synthesis report, this thing came up that the synthesis tool was replacing the floating point models with black boxes. It was further verified by using one floating point model in a very simple RTL, which was later synthesized and simulated at gate level.

The synthesis tool was actually synthesizing those floating point models which were provided by the vendor only for functional simulation; this was the reason of replacing the floating point models with the black boxes. The vendor instructs to use the NGC files provided for these models at synthesis time. Now there were
many NGC files, one NGC file for the design and rest are for the floating point models. The solution to this was to combine all these NGC files together and run the NGD simulation, which contains true timing information.

5.6.3 NGD simulation

The NGD gate-level simulation is used when it is not possible to simulate the direct output of the synthesis tool. This occurs when the tool cannot write UniSim-compatible VHDL or Verilog netlists. This process uses the post-place and route simulation model (a structural SIMPRIM-based VHDL or Verilog file) and a standard delay format (SDF) file generated by NetGen. The SDF file contains true timing information of the design [24].

Like the NGC simulation, this simulation allows to verify that the design has been synthesized correctly, and one can begin to identify any differences due to the lower level of abstraction. Instead of simulating the whole system, this time, the author first simulated the SIMD unit separately. While doing the gate level simulation of the SIMD unit, the simulation tool complained about the multiple accesses to the single port RAM of the register file. Actually, the synthesis tool was making a block RAM for the register file; this thing was resolved by forcefully instructing the synthesis tool to make it as a distributed RAM. The gate level simulation of the SIMD unit started giving the correct results. The following example shows how to instruct the synthesis tool for making a distributed RAM.

---

**Example 5.1: Distributed RAM style**

(* ram_style = 'distributed' *) reg [0:159] rf[0:31];

---

After performing gate level simulation on the SIMD unit separately, the Senior was attached with it and performed the NGD simulation; which was successful. Finally, the NGD simulation was performed on the whole system and it was also successful. If the NGD simulation is passed, then there are very few chances of the RTL failing on FPGA or ASIC.

The whole system was synthesized again; the .bit file was downloaded into the FPGA and tested. It was weird, because it was failing again. At this point, the only option left was to capture the on-chip signals at real time.

5.6.4 Debugging with ChipScope Analyzer

Theoretically, all signals (wires) on a traditional circuit board can be monitored using a probe, an oscilloscope, or a logic analyzer. The internal state of devices e.g. registers or buses are of course not possible to monitor using typical logic analyzer. This is where ChipScope comes in handy, provided by Xilinx. The ChipScope is a set of tools made by Xilinx that allows to easily probing the internal signals of
design inside an FPGA, much as one’s does with a logic analyzer. For example, while one’s design is running on the FPGA, one can trigger when certain events take place and view any of the design’s internal signals [25].

It has two main parts, one is embeddable IP cores that capture and store values of signals within the FPGA, like an “embedded” logic analyzer; the other is software tool that allows one to read the captured data and visualize it on a host computer, like the “screen” and “control panel” of a logic analyzer.

The designer connects signals of interest to a ChipScope ILA (Integrated Logic Analyzer) core, and then connects the ILA to a ChipScope ICON core. The ICON allows a host computer to access the data stored in the ILA. An ICON core is needed in all ChipScope designs. The ChipScope Analyzer tool is used to monitor the hardware signals connected to the ILA on host computer. The ChipScope system setup can be seen in Fig: 5.5 “Image from [26] (C) 2007 <Jason Agron>, Reprinted with permission”

![ChipScope system setup diagram](image)

**Figure 5.5.** ChipScope system setup.

The setup was arranged accordingly and tested, after examining the real time signals, this thing came up that the Senior was not providing the right instructions to the SIMD unit. It was occurring because of not executing the JUMP instruction, and the Senior program was running state away from top to bottom without taking any jump. This means that, the timing constraints in the NGD simulation did not allow to figure out why the Senior did not behave in the correct way when using JUMPs, even though the post NGD simulation worked correctly. In order to by pass this thing and test the system, the author initialized at start up the program and data of the SIMD unit into program and data memory respectively; after this a very simple program was written which was not having any JUMP instruction and ask the SIMD unit to invert the matrix. This program verified that the system is running properly.
Chapter 6

Results

The project has been finished with success. All the intended tasks have been completed successfully and the performance is satisfactory, if considering acceptable accuracy for the intended application.

- Verified the SIMD unit with all the combinations of instructions, and fixed the found bugs.

- The efficient matrix inversion algorithm is implemented, optimized and tested on the SIMD unit. The optimized algorithm consists of 118 instructions and it takes 152 clock cycles to complete one $4\times4$ matrix inversion.

- Singular matrices cannot be inverted by this algorithm, because the division $1/x$ fails when $x$ approaches zero.

- Accuracy can be affected on arrival of badly conditioned matrices, which causes very small or very large intermediate values. (Inspired by [10]).

- Integrated the SIMD unit with the Senior processor and wrote a simple firmware for the Senior for data upload/download through I/O ports.

- Wrote a firmware for the Senior for performing data upload/download into the SIMD unit through the DMA.

- Synthesized the Senior and the SIMD unit all together at 25 MHz with speed grade 10 and tested on Virtex-4 FPGA. This frequency was chosen for keeping single clock domain both in the SIMD unit and the Senior. It is possible to synthesize the SIMD unit at higher frequency than 25 MHz.
Chapter 7

Conclusions

7.1 SIMD processor

The SIMD processor is simple but mature enough to be used for MIMO processing especially in this matrix inversion algorithm. To use it for other operations may have some limitations, like relative memory address can only move within 64 locations.

The $1/x$ operation of the SIMD unit is very sensitive and could be hazardous depending upon the accuracy of target application. This is the result of considering $+0$ and $-0$ separately. Inverting $+0$ gives the value nearest to $+\infty$ that can be represented, and is saturated to $0x7FFF$ in fixed point. Similarly, inverting $-0$ gives the value nearest to $-\infty$ that can be represented, and is saturated to $0x8000$ in fixed point.

Some other aspects like data memory upload/download and changing the data memory pointer need to be set manually in the simulator. Unlike the RTL, the simulator always converts the input fixed point data into floating point and output floating point data into fixed point.

The output file of the program generated by the assembler can be used by testbench, but not the simulator. There is a script that converts the output file into purely $hex$ file understandable by the simulator. All of these operations need to be performed manually. A suitable future work might consider to perform all these operations automatically.

7.2 Verification and Matrix Inversion

All the instructions and their possible combinations work with corner cases, whether to keep an instruction in the left side instruction category or in the right side instruction category is a question difficult to answer; as it depends upon the op-
eration. There might be an operation which keeps the left side pipeline busy all the time and the right side pipeline free most of the time and vice versa.

The matrix inversion firmware could be improved further, if the STL/STG were right side instructions. Currently, they are left side instructions occurring at the end of the program and their right side pipeline is totally free. The second bottleneck in the performance of the matrix inversion is $1/x$ operation which takes 10 clock cycles. Though, in this case, it has been catered to a certain limit by overlapping the instructions, but for general usage of the SIMD unit, the hardware should be modified so as to reduce this inversion time.

### 7.3 Senior Integration

Using the SIMD unit as a target device for complex matrix inversion enhances the performance. It reduces the load on the main processor but the main processor has to keep track of the data going or coming from the SIMD unit. The communication between the Senior and the SIMD unit is performed both by I/O and the DMA. The I/O communication is not robust, because the SIMD unit sends data on every clock cycle, which is not possible for the Senior to receive. The reason being that, the SIMD unit sends data on every clock cycle while the Senior has to save that data after every read from the I/O port. However, the communication through the DMA is robust and fast.
Chapter 8

Future Work

There are few things which can be done to improve the system.

- If the last instruction of program stalls then there is need to put NOP instruction at the end of program to bypass this bug. This can be fixed for optimizing programs.

- The $1/x$ computation takes too much time i.e. 10 clock cycles, which is the main bottleneck of the architecture. The hardware should be modified so as to reduce this inversion time.

- The STL/STG are left side instructions occurring at the end of algorithm; they do not contain any instruction on their right side. If STL/STG were right side instructions things would improve because they can be merged with above instructions for improving the algorithm. (Inspired by [10]).

- Need to debug the system for investigating why the JUMP instruction is not executing in the Senior.

- It can be interesting to investigate the accuracy when badly conditioned matrices come, here badly conditioned means the matrices which have very small or very large intermediate values. (Inspired by [10]).

- The system can be synthesized on ASIC as well.
Appendix A

Instruction Set Reference

This chapter contains the instruction set reference.

Notations

\(\sim\) Negate imaginary part.
\(-\sim\) Negate real part.
- Negate both real and imaginary part.
acc An arbitrary accumulator register.
rA,rB An arbitrary general purpose register used as a source operand.
rD An arbitrary general purpose register used as a destination register.
sp Special purpose register.
offs Memory offset.

A complete list of the SIMD instructions with syntax and their description is given on next page.
## A.1 Left Side Instructions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td>No operation</td>
<td>This command does not do any useful work.</td>
</tr>
<tr>
<td>move acc,-rA</td>
<td>-rA → acc</td>
<td>Register move to accumulator by first changing operand’s signs.</td>
</tr>
<tr>
<td>move sp,-rA</td>
<td>-rA → sp</td>
<td>Register move to special purpose register by first changing operand’s signs for reciprocal operation.</td>
</tr>
<tr>
<td>move sp,-acc</td>
<td>-acc → sp</td>
<td>Accumulator move to special purpose register by first changing operand’s signs for reciprocal operation.</td>
</tr>
<tr>
<td>stl offs,-rA</td>
<td>-rA → (data_ptr + signed_offset)</td>
<td>Register store to memory locally by first changing operand’s signs. Relative address will be generated by adding the offset with data memory pointer.</td>
</tr>
<tr>
<td>stg offs,-rA</td>
<td>-rA → (0 + unsigned_offset)</td>
<td>Register store to memory globally by first changing operand’s signs. The offset address will be taken as an absolute address.</td>
</tr>
<tr>
<td>mul acc,~rA,-rB</td>
<td>~rA · -rB → acc</td>
<td>Floating point multiplication of both registers and stores the result in an accumulator.</td>
</tr>
<tr>
<td>mul acc,~acc,-rA</td>
<td>~acc · -rA → acc</td>
<td>Floating point multiplication of a register and an accumulator and then stores the result in an accumulator.</td>
</tr>
<tr>
<td>add acc,~rA,-rB</td>
<td>~rA + -rB → acc</td>
<td>Floating point addition of both registers and then stores the result in an accumulator.</td>
</tr>
<tr>
<td>add acc,~acc,-rA</td>
<td>~acc + -rA → acc</td>
<td>Floating point addition of a register and an accumulator and then stores the result in an accumulator.</td>
</tr>
<tr>
<td>mac acc,~rA,-rB</td>
<td>(~rA · -rB) + acc → acc</td>
<td>Floating point multiply and accumulate and then stores the result in an accumulator.</td>
</tr>
</tbody>
</table>

Table A.1. Left side instructions.
## A.2 Right Side Instructions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td>No operation</td>
<td>This command does not do any useful work.</td>
</tr>
<tr>
<td>ldl rD,offs</td>
<td>(data_ptr + signed_offset) → rD</td>
<td>Load contents of local memory into register. Relative address will be generated by adding the offset with data memory pointer.</td>
</tr>
<tr>
<td>ldg rD,offs</td>
<td>(0 + unsigned_offset) → rD</td>
<td>Load contents of global memory into register. The offset address will be taken as an absolute address.</td>
</tr>
<tr>
<td>move rD,~acc</td>
<td>~acc → rD</td>
<td>Accumulator move into general purpose register by first changing its operand’s signs.</td>
</tr>
<tr>
<td>move rD,sp</td>
<td>sp → rD</td>
<td>Read inverted value from special register into general purpose register.</td>
</tr>
</tbody>
</table>

Table A.2. Right side instructions.
Appendix B

4x4 Matrix Inversion Matlab Code

The 4×4 matrix inversion is first simulated on Matlab and then this code is used as a reference for writing the firmware for the SIMD unit. This Matlab source code was provided by the designer of the SIMD unit.

function R = inv4×4(M);
    % function for computing inverse of 4×4 matrix
    if (size(M, 1) ~= 4) || (size(M, 2) ~= 4)
        error('input matrix must be 4×4');
    end
    A = M(1:2, 1:2);
    B = M(1:2, 3:4);
    C = M(3:4, 1:2);
    D = M(3:4, 3:4);
    W = inv2×2(A);
    X = C·W;
    Y = inv2×2(D-X·B);
    Z = W·B·Y;
    R = [ W+Z·X, -Z ; -Y·X, Y ];
function R = inv2×2(M)
% helper function for inv4×4

    if (size(M, 1) == 2) || (size(M, 2) == 2)
        error(’input matrix must be 2×2’);
    end

    a = M(1,1);
    b = M(1,2);
    c = M(2,1);
    d = M(2,2);

    R = inv1x1(a*d-b*c) * [ d, -b ; -c, a ];

function R = inv1×1(M)
% helper function for inv4×4

    if ~isscalar(M)
        error(’input must be scalar’);
    end

    R = conj(M) * (1/real(M*conj(M)));
Bibliography


[5] Diagram of the execute states of the SIMD unit, Johan Eilert, unpublished

[6] Diagram of the register and accumulator file of the SIMD unit, Johan Eilert, unpublished


[9] Informal documentation of missing features of the SIMD simulator, Johan Eilert, unpublished

[10] Discussion by emails about the SIMD unit, Johan Eilert, unpublished


[12] Diagram of the multiply datapath of the SIMD unit, Johan Eilert, unpublished

[13] Diagram of the multiply-accumulate datapath of the SIMD unit, Johan Eilert, unpublished


Copyright

Svenska
Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping universitet Electronic Press se förlagets hemsida:
http://www.ep.liu.se/

English
The publishers will keep this document online on the Internet - or its possible replacement - for a period of at least 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping university Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page:
http://www.ep.liu.se/

© Sajid Hussain
Linköping, 2010