Norwegian University of
Science and Technology

An FPGA-based implementation of the Conjugate Gradient Method used to solve Large Dense Systems of Linear Equations

Torstein Habbestad

Master of Science in Electronics
Submission date: August 2011
Supervisor: Bjørn B. Larsen, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Abstract

To find the solution to large dense systems have always been a very time consuming problem, this thesis tries to accelerate this problem by implementing an highly pipelined conjugate gradient method on an FPGA, it has been used to solve dense systems of linear equations and has been tested and compared to a software version of the algorithm. The FPGA where capable of utilizing 90 % of the available memory bandwidth, in addition it is shown that the FPGA implemented Conjugate Gradient Method can be 30x faster compared to a custom made Conjugate Gradient method in software.

Acknowledgments

I like to say thank you to LimSim and Espen Øybø for introducing me to this project and all the help I got when I started on it, And I like to say thank you to my parents for the support they have shown while I have worked on the thesis.

Contents

I Introduction **8**

II Theory **10**

1 Algorithms that finds the solutions to systems of linear equations **10**

2 Conjugate Gradient Method **11**
2.1 Conjugate Gradient functionality 12
2.2 The Conjugate Gradient Method example 14

3 Convergence and Preconditioning **15**
3.1 Jacobi Preconditioner (diagonal preconditioner) 16
3.2 Incomplete Cholesky Factorization 18

4 Problems and advantages with FPGAs compared to CPUs **19**

5 Precision **19**

6 Representing numbers on computers **20**
6.1 The IEEE-754 standard for floating point arithmetic 20
6.1.1 Rounding modes 22
6.1.2 Exceptions 22
6.1.3 Floating Point Summation/Subtraction 22
6.1.4 Floating point multiplication 23

7 Round off error **23**

8 Introducing the Altera Floating Point Modules **25**
8.0.5 The Adder/Subtractor 26
8.0.6 Floating point multiplication 27
8.0.7 Floating Point Division 28

9 DDR-Memory **28**
9.1 DDR1 29
9.2 DDR2 29
9.3 DDR3 29

III Tools and programs used **30**

IV Architecture **31**

10 Design goals **31**

11 The memory module **31**

12 The Dot-Product Module **32**
12.1 The dot product control module 35

13 The vector-update pipeline **37**

14 Matrix-vector calculation **39**

15 The Top module **44**

V Testing and verification **50**

VI Results **54**

16 External memory bandwidth needed **54**

VII Conclusion **58**

17 Further work **58**

VIII Appendix A **61**

A PC-Specifications **61**

List of Figures

1 Single Precision format 21
2 Double precision format 21
3 Floating Point Addition 23
4 Addition of two numbers 24
5 Memory Block 32
6 This is the dot product module 33
7 Dot-Product, State-machine 35
8 Reduction circuit example 36
9 How the update of the vectors have been done 38
10 The Add Multiply Pipeline 39
11 The adder tree 41
12 The Accumulator 41
13 Reduction Circuit 42
14 The Matrix-vector multiplier 43
15 Divider Circuit 45
16 Top module control 47
17 Test control machine 51
18 Test System 52
19 Memory bandwidth efficiency 55

List of Algorithms

1 Conjugate Gradient Method 14
2 Conjugate Gradient Method with a preconditioner 17
3 Kahan Summation Algorithm 25

List of Tables

1 Direct Methods 10
2 Stationary iterative methods 11
3 Non-stationary iterative methods 11
4 The variables used in the Conjugate Gradient Method 13
5 Conjugate Gradient Method Example 15
6 List of preconditions 16
7 Floating point Special Numbers 26
8 Floating Point Addition module 27
9 Floating point multiplication module 27
10 Floating point division module 28
11 Arria II Resources 31
12 Resources used by the dot-product module 34
13 Similar operations 37
14 Resources used by the Matrix vector multiplier 42
15 Variable names 47
16 Theoretical iteration time 49
17 File format 50
18 Max size effect frequency 54
19 Total resource usage by the design 54
20 CPU and FPGA run-time comparison 57

Part I
Introduction

Field programmable gate arrays have traditionally been used to accelerate digital signal processing(DSP) applications and other applications where floating point arithmetic where not needed. However the logic density on FPGAs have now become very high, in addition, FPGA vendors like Altera and Xilinx have started to incorporate hard multipliers and memory-blocks directly into the logic fabric of their FPGAs. This has made it possible for researchers to study the possibility of making and implementing floating point arithmetic modules in FPGAs, many different attempts to do this can be read about in [2, 4, 3, 8, 13]. Moreover, FPGA vendors have also made floating point arithmetic IPs that are optimized for their FPGA architectures. As a consequence applications that traditionally have been implemented in software, for instance scientific computing that require floating point arithmetic have become available for implementation on FPGAs. A problem that can be found in many different areas of scientific computing is to find the solution to systems of linear equations.

Conjugate Gradient Method is one of the most used iterative methods to solve systems of linear equations. It is very effective on sparse systems of linear equations where direct methods are not applicable. However it can also be very effective on large dense systems of linear equations. Dense systems of linear equations can come from many different algorithms for instance, boundary element method [12], semi-definite programming and support vector machines [9], these like most other scientific problems demands high precision floating point arithmetic to be able to be solved. Because of this they have traditionally been implemented in software and solved by CPUs because of the high precision numbers they have supported for many years and because they are very easy to program compared to FPGAs. However operations that the Conjugate Gradient Method are using, like matrix-vector multiplication and dot-product calculations have been showed by Underwood [15] to be much more efficiently implemented in FPGAs, he also predict that they will be able to be one order of magnitude more efficient in BLAS operation than CPUs in 2010.

In [10] they have implemented a highly pipelined version of Conjugate Gradient Method used to solve dense systems of linear equations. They were able to get a very high performance because they stored the matrix in the logic fabric of the FPGA instead of storing it in external memory, as a consequence of this they were only able to solve small matrices. A Gaussian implementation have been studied in [16], this is however also only capable of solving small matrices.

This thesis will continue on the work done in [5], in that study the Conjugate Gradient Method where implemented in SystemC to test if it would be effective to implement the Conjugate Gradient method on an FPGA, it was found that as long as the matrix-vector multiplication was solved efficiently and the external memory where read large chunks at a time it was possible to accelerate the Conjugate Gradient Method using an FPGA. In this thesis the Conjugate Gra-

dient Method will be implemented on an FPGA and tested on larger matrices that was not possible to do in the preliminary study. The possibility of using Preconditioning to accelerate the algorithm even more will also be studied.

Part II
Theory

1 Algorithms that finds the solutions to systems of linear equations

Finding the solution to systems of linear equations have always been an important problem in mathematics, hence many different methods have been developed. These will be mentioned in this section. There are two categories of algorithms that can be used to find the solution to systems of linear equations, the older direct methods and the newer iterative methods that are more suitable to be implemented on a computer.

Direct methods are methods that finds the solution to systems of linear equations by decomposing the matrix into a canonical form which can then be solved easier compared to solving the system with the original matrix. They are able to find the solution in a finite number of steps, but the speed of decomposing the matrix is very dependent on the size of the matrix, hence they are very inefficient on large matrices.

Examples of direct methods can be seen in table 1. A Gaussian elimination algorithm have been implemented on FPGA and can be seen in [16].

Direct methods
Gaussian Elimination
LU Decomposition
Cholesky decomposition
QR-decomposition

Table 1: Direct Methods

The iterative methods can further be divided into two categories the stationary iterative methods and the non-stationary iterative methods.

The stationary iterative methods can be expressed as $x_{(i)} = Ax_{(i-1)} + c$, where neither A nor c depends on the iteration i. These methods usually converge much slower than the non-stationary methods, but can be used as preconditioners to the more advanced non-stationary methods. Different stationary iterative methods can be seen in table 2. The Jacobi method have been implemented in an FPGA and can be seen in [11].

Stationary iterative methods
Jacobi method
The Gauss-Seidel method
The successive over-relaxation method (SOR)
The symmetric Successive Over-relaxation method (SSOR)

Table 2: Stationary iterative methods

The non-stationary iterative methods differ from the stationary iterative method in that they use information that changes in every iterations to compute the result. Most of these are based on finding and using sequences of orthogonal vectors to find the solution to the system of linear equations. These methods start by guessing the solution, this will be the first approximation, it then builds on this approximation in each iteration to make it closer and closer to the real solution. The efficiency of iterative solvers usually depends more on the condition number of the coefficient matrix than on the size of the matrix, this makes them very attractive to use on large matrices which will be to large for the direct method to solve efficiently.

Some example of non-stationary iterative methods can be seen in table 3:

Non-stationary Iterative Methods
Conjugate Gradient Method (CG)
Minimum Residual (MINRES)
Conjugate Gradient Squared (CGS)
BiConjugate Gradient (BiCG)

Table 3: Non-stationary iterative methods

More information about the different algorithms can be seen in [1]. The Conjugate Gradient method is an iterative non-stationary method as can be seen in table 3. This algorithm will be explained in more detail in the next section.

2 Conjugate Gradient Method

Conjugate Gradient method was first proposed by Hestenes and Stiefel in [6]. It has become one of the most popular and well known iterative methods for solving systems of linear equations. It is mostly used for very large sparse matrices where it is not possible to use a direct methods like for instance Gaussian elimination, but it can also be very effective for large dense systems.

The system of linear equations can be written in the matrix form: $Ax = b$, where x is the unknown vector, b is the known vector and A is the coefficient matrix . The algorithm works by constructing a set of mutually conjugate search directions, hence the name Conjugate Gradient method. To be certain

that the Conjugate Gradient Method will work, the matrix A need to be square, symmetric and positive definite as defined under.

- $A = A^T$,
- $x^T Ax > 0$ for all nonzero vectors x.

Dense systems comes into play in many important settings, for instance, structural analysis, semi-definite programming , support vector machines and boundary element formulations typically give rise to fully populated matrices. In the Conjugate Gradient method the start point $x_{(0)}$ can be chosen arbitrarily, but in the solution presented here it will always start in the origin, this means that all elements in the x vector starts at zero. The Conjugate Gradient Method have some big advantages compared to other methods, for instance, it only needs to store 5 vectors that can be updated and replaced in every iteration. Another big advantage is that it only needs to perform one matrix-vector multiplication every iteration, this will save alot of time since this is the most time consuming operation in the algorithm. The Conjugate Gradient Method will in theory find the solution x in less than N iterations, where N is the number of rows in A. However, accumulated floating point round off error causes the residual to gradually lose accuracy, and cancellation error causes the search direction to lose A-orthogonality, hence, this does not always happen in practice.

2.1 Conjugate Gradient functionality

The Conjugate Gradient Method starts by setting $r_{(0)} = b - Ax_{(0)}$, if x starts in the origin this will simplify to $r_{(0)} = b$, the algorithm then proceeds to find the approximate solution to the problem in an iterative way by constructing iterates of the form $x_{(i+1)} = x_{(i)} + \alpha d_{(i)}$, where $d_{(0)} ... d_{(n)}$ is the search direction vectors. The search directions are constructed to be mutually conjugate to each other to satisfy the conjugacy property $d_{(i)}^T Ad_{(j)} = 0$, where $i \neq j$. α can be thought of as the step-length taken in the current search direction $d_{(i)}$. α is calculated by $\alpha = \frac{r_{i+1)}^T r_{(i)}}{d_{(i)}^T Ad_{(i)}}$. The residual is updated by subtracting αAd from the last residual, this will make it orthogonal to the last residual. The residual can be updated by either using the simple recurrence $r = r - \alpha q$ or a matrix-vector multiplication $r = b - Ax$, because of this choice the Conjugate Gradient Method became very popular since it only needs one matrix-vector multiplication in each iteration. The disadvantage of using the recurrence is that the residual is generated without any feedback from x, this can cause the algorithm to converge to a point near x because of accumulated round off error. By using the matrix-vector multiplication instead of the recurrence once in a while it is possible to get rid off the accumulated round off error. In [14] it is recommended to use it every $\sqrt{N}$ iteration. The residual indicates how far we are from the correct value of b, the smaller the residual is the closer we are to the correct solution. This can easily be seen by looking on the formula for the residual $r = b - Ax$, if r is zero then $Ax = b$, hence the solution has been found.

β is the Gram-Schmidt constants that are used to make the search direction A-orthogonal to all the past search directions. β is constructed by the ratio of the current and last residual, $\beta = \frac{r_{(i+1)}^T r_{(i+1)}}{r_{(i)}^T r_{(i)}}$.

It is normal to stop when the norm of the residual, δ_{New} , falls below a specified value often this value is some small fraction of the initial norm of the residual, $\| r_{(i)} \| < \varepsilon^2 \| r_{(0)} \|$.

The different variables that the Conjugate Gradient Method use has been summarized in table 4.

Variable	Type	Description
A	Matrix	The coefficient matrix
b	Vector	The known vector
x	Vector	The unknown vector
r	Vector	The residual
d	Vector	The search direction
q	Vector	The result of the Matrix-vector multiply
α	Scalar	Step-length
β	Scalar	Gram-Schmidt constants that is used to find the next A-orthogonal search direction
ε	Scalar	The error tolerance

Table 4: The variables used in the Conjugate Gradient Method

The complete algorithm for the conjugate gradient method can be seen in algorithm 1.

Algorithm 1 Conjugate Gradient Method

$x_{(0)} = 0$

$r_{(0)} = b$

$d_{(0)} = r_{(0)}$

$\delta_{New} = r_{(0)}^T r_{(0)}$

$\delta_{First} = \delta_{New}$

While$(\delta_{New} > \varepsilon^2 \delta_{First})$ do

$q_{(i)} = Ad_{(i)}$

$\alpha = \frac{\delta_{New}}{d_{(i)}^T q_{(i)}}$

$x_{(i+1)} = x_{(i)} + \alpha d_{(i)}$

$r_{(i+1)} = r_{(i)} - \alpha q_{(i)}$

$\delta_{Old} = \delta_{New}$

$\delta_{New} = r_{(i+1)}^T r_{(i+1)}$

$\beta = \frac{\delta_{New}}{\delta_{Old}}$

$d_{(i+1)} = r_{(i+1)} + \beta d_{(i)}$

2.2 The Conjugate Gradient Method example

A simple example of the algorithm is provided to help the reader get a good understanding of how the different variables in the Conjugate Gradient Method develop over the iterations, the results can be seen in table 5. The simple example system of linear equation used is: $\begin{bmatrix} 3 & 2 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$

Variable	Initialization	Iteration 1	Iteration 2
q	$[0,0]^T$	$[17,22]^T$	$\left[-8.3\times10^{-2}, 6.2\times10^{-2}\right]^T$
d^Tq	0	139	7.4×10^{-3}
α	0	0.18	0.7
δ_{Old}	0	25	5.2×10^{-3}
δ_{New}	25	5.2×10^{-3}	10^{-31}
r	$[3,4]^T$	$\left[-5.8\times10^{-2}, 4.3\times10^{-2}\right]^T$	$\left[10^{-16}, 10^{-16}\right]^T$
x	$[0,0]^T$	$[0.54, 0.72]^T$	$[0.5, 0.75]^T$
β	0	2.1×10^{-4}	10^{-28}
d	$[3,4]^T$	$\left[-5.7\times10^{-2}, 4.4\times10^{-2}\right]^T$	$\left[10^{-16}, 10^{-16}\right]^T$

Table 5: Conjugate Gradient Method Example

As can be seen in the example, the answer x is approximately correct in the first iteration and then in the second iteration it is correct, this is one of the big advantages of Conjugate Gradient method, it can be stopped before the exact answer is found if an approximate answer is good enough. Another important thing that can be seen in the example is that the variables becomes very small when the algorithm is close to the solution, hence it is important to stop the algorithm in time before the variables start to underflow, or before any variable becomes zero, if a variable becomes zero it is possible that a divide by zero happens. A very good explanation of the Conjugate Gradient Method can be found here [14].

3 Convergence and Preconditioning

The convergence factor of the conjugate gradient method, meaning its ability and efficiency to converge to the real value of x depends to a large extent on the condition number of the coefficient matrix A. If the matrix is symmetric the condition number will be the ratio of the largest and smallest eigenvalue of the coefficient matrix, $\kappa = \frac{\lambda_{Max}}{\lambda_{Min}}$, furthermore, for symmetric positive definite matrices all the eigenvalues will be positive. If the condition number is small the matrix is said to be well-conditioned and will most likely converge to an approximate value of x very quickly, on the other hand, if the condition number is large the matrix is said to be ill-conditioned and will most likely converge very slowly, in fact, sometimes it will even fail to converge. Furthermore, the Conjugate Gradient Method will converge faster if the eigenvalues of the coefficient matrix are clustered together instead of being irregularly distributed between λ_{Max} and λ_{Min}. The condition number can be though of as a number that reflect how sensitive the result will be to small perturbation in either b or A. If the condition number is large small round off errors can result in a large

Preconditioners
Quasi-Newton
Incomplete LU factorization
Incomplete Chomsky Factorization
Jacobi
Symmetric Successive Over Relaxation(SS OR)

Table 6: List of preconditions

error in x.

If the matrix is ill-conditioned it can be preconditioned to reduce the condition number. If we found a matrix M that approximated A, but was easier to invert. Then we could solve the system $Ax = b$ indirectly by solving the system $M^{-1}Ax = M^{-1}b$, if the condition number of $M^{-1}A$ were smaller than the condition number of A or if the eigenvalues of $M^{-1}A$ were clustered together more than those of A, then we could solve the system $M^{-1}Ax = M^{-1}b$ in a fewer number of iterations compared to the system $Ax = b$. The efficiency of the different preconditions available depends very much on the structure of the coefficient matrix, hence, the structure of the matrix needs to be very well known to be able to make an efficient preconditioner matrix, in addition, if the matrix A is changed the precondition matrix needs to be recalculated as well. The preconditioned Conjugate Gradient Method can be seen in algorithm 2.

In addition to the preconditioners ability to improve the convergence rate, some other factors needs to be taken into consideration when choosing the preconditioner to be used on the matrix, for instance the extra storage needed to implement the preconditioner, the time it takes to construct the preconditioner matrix and the extra time it takes to apply the preconditioner in every iteration. Different types of preconditioners can be seen in table 6.

The Jacobi and Incomplete Chomsky decomposition preconditioner will be discussed in more detail.

3.1 Jacobi Preconditioner (diagonal preconditioner)

The Jacobi Preconditioner is the simplest preconditioner that can be used. It consist of the diagonal element of A on its diagonal and zeros everywhere else.

$$M_{ij} = \begin{cases} A_{ii} & i = j \\ 0 & otherwise \end{cases}$$

To invert a diagonal matrix is very simple, this is done by inverting the element on the diagonal, for instance if the matrix have a 2 on its diagonal it will be $\frac{1}{2}$ in the inverted matrix, all the other elements will be zero.

$$M_{ij}^{-1} = \begin{cases} \frac{1}{a_{ij}} & i = j \\ 0 & otherwise \end{cases}$$

The Jacobi preconditioner is very effective on diagonal dominant matrices. Because all except the diagonal element is zero, it only needs $64N$ number

Algorithm 2 Conjugate Gradient Method with a preconditioner

$x_{(0)} = 0$

$r_{(0)} = b$

$s_{(0)} = M^{-1} r_{(0)}$

$d_{(0)} = s_{(0)}$

$\delta_{New} = s_{(0)}^T r_{(0)}$

$\delta_{First} = \delta_{New}$

$While(\delta_{New} > \varepsilon^2 \delta_{First})\ \ do$

$q_{(i)} = A d_{(i)}$

$\alpha = \frac{\delta_{New}}{d_{(i)}^T q_{(i)}}$

$x_{(i+1)} = x_{(i)} + \alpha d_{(i)}$

$r_{(i+1)} = r_{(i)} - \alpha q_{(i)}$

$s_{(i+1)} = M^{-1} r_{(i+1)}$

$\delta_{Old} = \delta_{New}$

$\delta_{New} = s_{(i+1)}^T r_{(i+1)}$

$\beta = \frac{\delta_{New}}{\delta_{Old}}$

$d_{(i+1)} = s_{(i)} + \beta d_{(i)}$

of bits to store the preconditioner and $M^{-1}r$ can be calculated by doing N multiplications. This makes the Jacobi preconditioner a very memory efficient and a computationally efficient method that will use up very little extra time on each iteration, only N clock cycles extra. Of course since it is so simple it has some downsides. It is only a mediocre preconditioner compared to more advanced methods, like for instance Cholesky factorization.

The Jacobi preconditioner do not need much extra resources to be implemented, it needs to be able to store the reciprocals of the diagonal of the matrix A in a memory called M^{-1}, since all other elements are zero they do not need to be stored and the preconditioner will effectively use the same memory-space as a vector, it also needs extra memory for the s vector, extra memory bits needed will be: $64 \times 2N = 1048576$ bits, this will easily fit on the Arria II FPGA.

The preconditioned Conjugate Gradient Method will need to use some extra time in the initialization stage, it need to divide all the elements on the diagonal in A, this will use N+24 clock cycles, it can use the same divider that are already implemented in the circuit. In addition, it need to calculate the product $s = M^{-1}r$, since a diagonally matrix is basically a vector only a simple multiplier is needed to do this, clock cycles needed is N+11, this operation can start when the first element of r has been calculated and work in parallel with it for the rest of the elements. Hence, the Conjugate Gradient Method preconditioned with Jacobi only needs $A_{Lat} + M_{Lat} + 2$ extra clock cycles extra every iteration compared to normal Conjugate Gradient Method.

3.2 Incomplete Cholesky Factorization

Jacobi preconditioner sometimes improve the convergence rate only marginally compared to other methods. To get a better convergence rate it is possible to use a more advanced type of preconditioning, for instance the incomplete Cholesky factorization, this method decompose the matrix A into LL^T where L is a lower triangular matrix and $L^T$is the transpose of the lower triangular matrix. This will however produce a dense preconditioner matrix instead of the diagonal matrix the Jacobi preconditioner introduced. This means that a new matrix-vector multiplication needs to be performed every iteration. Since this operation is responsible for most of the time consumed in one iteration the time required to do one iteration would practically double. Hence, if the Incomplete Cholesky Factorization is going to reduce the total run time of the algorithm compared to the un-preconditioned Conjugate Gradient Method it will need to converge in less than half the number of iterations compared to the un-preconditioned Conjugate Gradient Method. Since the preconditioner-matrix will be to large to store internally in the FPGA it needs to be read from the external memory. This means that the amount of external memory needs to be double that of the basic conjugate gradient. In [9] the effect of applying the Incomplete Cholesky Factorization on dense symmetric positive matrices have been studied, it was concluded that that the effect it had on the convergence rate was not large compared to diagonal preconditioning. It can be concluded that the effect different preconditioning technique have on the

convergence rate of the conjugate gradient method depends heavily on how the coefficient matrix look like, and it is impossible to find a preconditioner that are generally better than another for all types of coefficient matrices. It can however also be concluded that the preconditioner that will cause the smallest increase in iteration time and at the same time improve the convergence rate will be the Jacobi preconditioner.

4 Problems and advantages with FPGAs compared to CPUs

FPGAs have some advantages over traditional CPUs, they can for instance use parallelism and pipelining much easier, they are also good at specialized work for example as hardware accelerators for specialized algorithms. CPUs on the other hand has a much higher clock frequency, they are more general and can be used for almost anything. They are however not programmable, the hardware resources are fixed and its operation can only be changed by giving it instructions trough software on how to use those resources. FPGAs on the other hand can be re-programed as many times as needed when a fault or a way to make it more efficient is found it is easy to reprogram it.

When solving the Conjugate Gradient Method for dense systems it may not be an advantage to have a the high clock frequency because the memory are usually much slower and in most of the clock cycles it have to wait on data from the memory. An FPGA with a smaller clock frequency can still be faster than the CPU because in every clock cycle it will get data to work with and if it gets more than one element it can easily work on those in parallel. Because of the higher Clock frequency in CPUs they will use much more power than FPGAs but still be very inefficient because of all the idle cycles were it have to wait on data from memory.

The FPGA also have the advantaged that it can store much more in internal memory than CPUs. CPUs have cache but this is not very efficient if the data is not reused. The FPGAs have the advantage that it can be easily connected to more memory modules as long as the required I/O pins are there. The problems that FPGAs have to deal with is the low clock frequency and it usually have to be specialized for one assignment. In today's competitive marked FPGAs have the advantage of low time to marked time. There is FPGAs on the marked today that will be able to connect up to six ddr3 memory module, hence they will be able to get a memory bandwidth of up to 70GB/s.

5 Precision

Many scientific applications needs high dynamic range on the floating point calculations to maintain numerical stability. For instance in the Conjugate Gradient Method the residual needs to be orthogonal to each other and the search directions need to be conjugate to each other. Because of round off error it is

impossible to calculate these numbers exact, however, by using number systems with both high range and precision it is possible to minimize these problems. The use of reconfigurable hardware to perform high precision operation such as IEEE floating point operations has been limited in the past by FPGA resources. Reconfigurable logic has the potential to yield significant speedup for many of these algorithms, though traditionally it has been unable to handle IEEE floating-point formats. Fixed point can only be used if the dynamic range of the numbers are small. The IEEE standard for floating point numbers is the same standard that are implemented in most of the CPUs today.

6 Representing numbers on computers

Squeezing infinitely many real numbers into a finite number of bits requires an approximate representation. There are several ways of approximating real numbers on computers. For instance fixed point where a radix point is put somewhere in the middle of the digits, where the digits on the left of the radix point represent the fraction and the digits to the left represent the integer part of the number is one way of doing it. Examples of other ways to represent numbers on computers is binary-coded decimal and logarithmic number system. However it is very difficult to represent numbers that have a high dynamic range and at the same time a good precision with these number representations and this is usually needed when doing scientific calculations. This is why floating point numbers are usually used for these types of computations.

Floating point numbers are represented in scientific notation, it consist of a sign, a base number and an exponent, for instance the number $-240,45$ can be represented as $-2,4045 \times 10^2$, this makes it much easier to represent very high and very low numbers. However the high range comes at the expense of the precision, the floating point numbers are not able to get the same precision as fixed point numbers since some of the bits in the floating point format needs to be used to represent the exponent of the number. In numerical methods like for instance the Conjugate Gradient method both the range and the precision is of great importance and therefore the floating point format have to be used. The floating point standard that have become a de-facto standard is the IEEE-754 standard for floating point arithmetic.

6.1 The IEEE-754 standard for floating point arithmetic

The IEEE-754 standard defines the format of floating point numbers and how they should be stored in a computer, how the arithmetic operations on floating point numbers should be done, rounding modes, how to handle exceptions and conversion between different formats. This enables all machines following the standard to exchange data and to calculate the exact same result when doing floating point operations. Thus, when a program is moved from one machine to another, the results of the basic operations will be the same in every bit if both machines support the IEEE standard. This greatly simplifies the porting

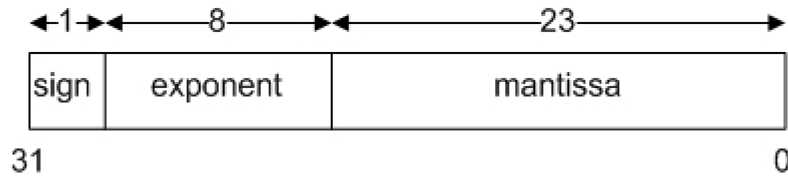

Figure 1: Single Precision format

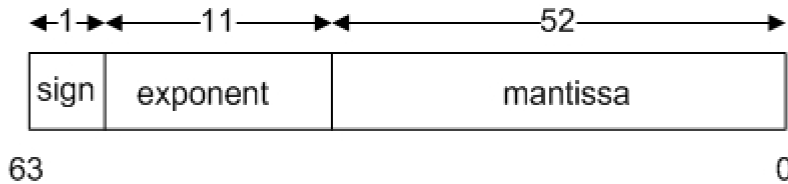

Figure 2: Double precision format

of programs.

The standard defines two different formats to store floating point numbers on computers : Single Precision as can be seen in figure 1, and double precision as can be seen in figure 2. Floating-point representations consist of a sign s, a base b, a mantissa m and an exponent e and can be represented as $(-1)^s \times m \times b^e$.

The IEEE Standard for floating point arithmetic is the most common representation for real numbers on computers.

The floating point formats is a bit-string that consist of three section, the most significant bit is the sign 0=+, 1=-, the next section is the exponent. The exponent is the number that decides the range of the floating point number specified in the format. This number is biased to be able to represent both positive and negative exponents without needing to store negative exponents. The remaining bits is the mantissa of the floating point number, this number decides the precision of the number.

The bias is used to be able to represent negative exponent without the need to store negative exponent. For instance a number with a true exponent of 2 will be stored as a number with an exponent of Bias + 2. The most negative exponent -1022 will be stored as Bias+(-1022) = 0.

In double precision the exponent occupy 11 bits, which means that it is able to represent numbers in the range 2^{-1022} to $(2 - 2^{-52}) \times 2^{1023}$, this is approximately $10^{-323,3}$ to $10^{308,3}$ in the decimal system. Of course the big range of floating point numbers comes at the expense of the precision, the precision of the number will not be as high as with fixed point format numbers. The double precision have 52 bits to represent the mantissa. All the arithmetic operations is not necessarily valid for floating point numbers. For instance the addition operation is not Associative nor commutative. Meaning that the order the values are added in can can effect the result.

6.1.1 Rounding modes

Given any fixed number of bits, most calculations with real numbers will produce quantities that cannot be exactly represented using that many bits. Therefore the result of a floating-point calculation must often be rounded to the closest representable number in order to fit back into its finite representation. The most common situation is illustrated by the decimal number 0.1. Although it has a finite decimal representation, in binary it has an infinite repeating representation. Thus when a base of 2 is used, the number 0.1 lies strictly between two floating-point numbers and is exactly representable by neither of them.

The IEEE-754 defines four rounding modes.

- Round to nearest, if ties it round to get a result with a LSB of 0.
- Round toward zero
- Round toward positive infinity
- Round to negative infinity

6.1.2 Exceptions

The IEEE-754 defines the following exceptions:

- Overflow: Happens when the exponent of the number is to large to be represented
- Underflow: Happens when the exponent of the number is to small to be represented
- Divide by zero
- Invalid operation
- Inexact

6.1.3 Floating Point Summation/Subtraction

This is the most complex arithmetic operation to be implemented. Floating point summation consist of several distinct steps: First the two numbers are compared to find the number that have the smallest exponent of the two. This number is then shifted to the rigth until the two numbers have the same exponent. The two numbers are then summed together to produce the un normalized result. The un normalized result is then normalized by shifting the result to the left until the most significant bit is equal to one. In the Altera floating Point IPs these operations are pipelined to achieve a high clock frequency and hence a high throughput is possible. These IPs are usually capable of handling a clock frequency of 150-250 MHz. Since floating point operation demand so much logic to be performed it will be impossible to do it in one clock cycle without getting

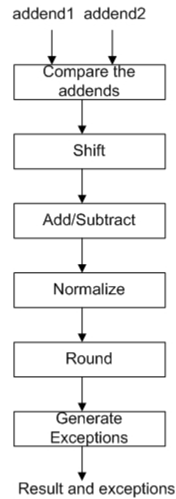

Figure 3: Floating Point Addition

a very low maximum clock-frequency. The research have focused on doing these operations in multiple steps in a pipeline. To be able to divide the logic into different clock cycles. This is done by inserting register between the different logical operation that the floating point operation consist of. Pipelining will be able to get a mush higher clock frequency and so will be able to achieve higher throughput if the number of operations to be performed is high. There will however be an obvious trade-off between the maximum-frequency and the area needed to implement the module.

6.1.4 Floating point multiplication

Floating point multiplication of two numbers that are represented in the IEEE 754 format are usually done in the following way. If the two input numbers are X and Y, then the sign of the result will be (the sign of X XOR the sign of Y), the mantissa of the result will be $Y.man \times X.man$ and the exponent of the result will be the exponent of X plus the exponent of Y. If the result is not normalized it need to be normalized before it is stored, and rounded if necessary. The multiplication operation are simpler than the addition because of the way the floating point numbers are represented.

7 Round off error

Round off error, how they come into play and how they can effect the final result of an arithmetic operation is something everyone that works with numerical methods need to be aware of, hence it will be presented here. Round off errors is created because of the fact that computers are only able to store numbers

with finite precision because of finite memory. Floating point numbers are finite-precision numbers that represent infinitely precise numbers, the smallest precision possible is known as the machine-accuracy, ε, this is the number that when added to one will produce a number different from one. This quantity depends on the size of the mantissa, in double precision the machine accuracy is $2^{-53} \approx 1.1 \times 10^{-16}$ and for single precision it is $2^{-24} \approx 6 \times 10^{-8}$.

Every arithmetic operation introduces an error into the result that are at least equal to the machine-accuracy. The round off error are cumulative meaning that the error will grow with the number of operations performed. Depending on the algorithm used it will grow between $\sqrt{n}\varepsilon$ and $n\varepsilon$, where n is the number of operations and ε is the machine precision. However, the round of error can sometimes become much larger, for instance, because of the way addition works in the floating point format, the mantissa to the smallest number are shifted to the Right as many times as the difference between the two numbers exponents. In extreme cases, if one of the numbers are much larger than the other the result of the addition will be equal to the largest number. How the digits are lost can be seen in figure 4, the values are presented in the decimal system for ease of understanding. It can be seen that the lower four digits are effectively lost and hence does not contribute to the result.

```
  1000.453
     0.2313462
---------------
= 1000.684
```

Figure 4: Addition of two numbers

This can become a large source of error in operations where many floating point numbers are added together, for instance in matrix-vector multiplications and dot-products because the numbers already accumulated will usually be much larger than the individual numbers coming in. There are however methods that can be used to solve this if it turns out to be a problem. One way is to ensure that the two numbers in the addition are relatively similar in size. This can be ensured by sorting the numbers from smallest to largest before the addition and then add the numbers in increasing order. Another way to solve the problem is to use the Kahan summation algorithm, how this algorithm work can be seen in algorithm 3.

Algorithm 3 Kahan Summation Algorithm

$sum = 0$
$c = 0$
loop
$y = input - c$
$t = sum + y$
$c = (t - sum) - y$
$sum = t$
end loop

By using this method the round off error would become independent on the number of operations done. Because it compensates for the error found in the last operation by subtracting it from the next number to be added.

However, both of these methods are relatively difficult to implement on an FPGA. The first method because of the sorting process needed and the Kahan Summation because the variables are dependent on each other in the algorithm which would make the operation very hard to pipeline. An easier way to reduce the round off error is to use a floating point format with a small machine precision, this is way the double precision number format will be used in all calculations.

The Conjugate Gradient Method can be effected by round off error in two ways, it can either make the search directions not exactly conjugate or it can make the residual not exactly orthogonal to the last residual.The conjugate gradient method needs the highest possible precision to make the algorithm converge. Even though in theory the algorithm will converge in less than the matrix order number of iterations it will not do this in practice because of the round of error in the calculations. It have to be seen if the round off error is accumulated in the algorithm, if so ways of improve the round off error needs to be looked in to. When using the Conjugate Gradient Method it is possible to calculate the exact residual by using an extra matrix-vector multiplication and storing the b vector in on-chip memory it is possible to calculate the exact value of the residual in every 100 or so iterations. By doing this the round off error that might have been accumulated in the residual will go away. This is done to not be completely depended on the recurrence that is usually done to calculate the residual. Because of the finite resolution the residual will never be exact orthogonal to each other, round off error is bound to accumulate. More about round off error can be seen in [7].

The floating point modules that is going to be used have been developed by altera and optimized for the Altera FPGAs.

8 Introducing the Altera Floating Point Modules

It has been very problematic in the past to implement a high precision floating point module on FPGA this has mainly bin because of little resource in logic and low clock frequency. However, there has recently been done much research

lately on this problem in for instance [2, 4, 3, 8, 13] . Here they have pipelined the typical stages of a floating point operation to be able to get a very impressive clock frequency. This has enabled designers in high performance computing to use FPGAs in scientific computing and solving problems that demand a high accuracy on the computations.

Altera has made floating point IP modules that incorporates most of the IEEE 754 standard. It incorporates single-precision, double-precision and single extended precision, in addition, special values like zero, infinity, denormal numbers, and NaN bit are also supported. How the bit-string will look like for special values can be seen in table 7.

Meaning	Sign	exponent	Mantissa
Zero	X	All '0'	All '0'
Positive Denormalized	0	All '0'	Non-zero
Negative Denormalized	1	All '0'	Non-zero
Positive Infinity	0	All '1'	All '0'
Negative Infinity	1	All '1'	All '0'
Not-a-Number(NaN)	X	All '1'	Non-zero

Table 7: Floating point Special Numbers

The floating point arithmetic modules have been specifically designed and optimized for the altera FPGAs. The IEEE-754 defines many different rounding modes, but the Altera modules only support the most commonly used rounding mode, which is the round-to-nearest-even mode. This rounding mode function like this, it rounds the result to the nearest floating point number, if it is exactly in the middle between two representable numbers it rounds the result so that the least significant bit becomes a zero, which is even, hence the name round-to-nearest-even-mode.

In Quartus the megawizard can be used to generate the arithmetic floating point modules, they are highly modifiable, for instance, it is possible to choose how many stages the floating point module is going to have in its pipeline, this will introduce an obvious trade-off between area and max frequency this have been illustrated in table 8, 9 and 10. These floating point modules from Altera makes it possible for designers to easily do floating point arithmetic's on FPGAs, since they have been optimized for Altera devices it is possible to get very impressive performance from them.

8.0.5 The Adder/Subtractor

The Adder/Subtractor is able to handle floating point summation and/or subtraction. It has two inputs for the addends and an output port for the result, the bus width depends on the format used, single precision gives 32 bit and double precision gives 64 bits. Furthermore, it have optional ports to handle exceptions, for example, overflow, underflow, zero and not a number(NaN). It have an optional clock enable and asynchronous clear port. If the clock enable

is high the data in the pipeline will move one pipeline-step forward every clock cycle, else it will stand still in the pipeline. If the asynchronous clear port is high all the pipeline stages and the result will be set to zero. The module also have an optional add_sub port, this port will make the module able to do both addition and subtraction depending on the value on the port, if it is one it will subtract and if it is 0 it will add. The module can be optimized for both speed and area. The options for the output latency are the same for both single and double precision. The choices of latency and the resource usage and max frequency for the addition module can be seen in table 8.

$Latency$	$f_{max}[MHz]$	$LUTs$	$Registers$	$Memory\ bits$
14	282	1467	1611	84
13	267	1551	1498	78
12	258	1542	1336	78
11	211	1393	1252	81
10	229	1343	1229	0
9	221	1400	1118	0
8	208	1387	977	0
7	198	1410	836	0

Table 8: Floating Point Addition module

8.0.6 Floating point multiplication

Floating point multiplication is a much simpler arithmetic operation to be implemented on an FPGA compared to floating point addition. This is mostly because of the way the floating point format is build up. The floating point format is divided into an exponent part, a mantissa part and a sign part. To do a multiplication the sign of the multiplicands is XOR-ed together to get the sign of the result. The exponents is added to each other to get the exponent of the result, and the mantissas is multiplied together with the help of the hard multipliers on the FPGA. As can be seen all these operations can be done in parallel, hence the pipeline for the multiplication module can be shorter than the pipeline in the addition module. Floating point multiplication usually demands very low area compared to other operations because of the dedicated multiplication circuits that most modern FPGAs have built in. The choices of latency and the resource usage and max frequency for the multiplication module can be seen in table 9.

$Latency$	$f_{max}[MHz]$	$LUTs$	$Registers$	$Memory\ bits$	$Multipliers$
11	251	466	603	108	10
10	231	435	581	75	10
6	175	429	476	0	10
5	173	408	371	0	10

Table 9: Floating point multiplication module

8.0.7 Floating Point Division

The floating point division module is the floating point module that have the lowest f_{Max}, in addition it is the most resource demanding module and is probably the module that will constrain the f_{Max} of the circuit . The module have been implemented on the Arria II to see how it performed, the result can be seen in table 10.

$Latency$	$f_{max}[MHz]$	$LUTs$	$Registers$	$Memory\ bits$	$Multipliers$
24	171	871	1306	6158	44
10	104	707	1099	4608	44

Table 10: Floating point division module

9 DDR-Memory

There are several memory types that can be interfaced by the Arria II FPGA, QDR II, RLDRAM, SRAM and DDR-SDRAM. Of all these options only the DDR-SDRAM is applicable to the needs of this design, because of the high storage requirements of the matrix A. DDR-DRAM is a type of memory that stores each bit in a separate capacitor, if the capacitor is charged it represent a 1 if it is discharged it will represent a 0. Because the capacitors gradually leak charge, the charge have to be refreshed periodically, this is done automatically by the controller every 7.4 micro second and takes about 18 clock cycles to do, this will make the maximum bandwidth lose about 2 %.

The main advantage the DDR-SDRAM have over other memory-types is its simplicity, it only need one transistor and one capacitor to store each bit, this makes it possible to reach high densities at a low cost. DDR memories use both the rising and falling clock edge to send data, this means that it can send 2*number of data bits on the memory-module/per clock cycle. The memory controller can both be run in half-rate mode and full-rate mode. In full-rate mode the memory-module will transfer 2*Data_bits every clock cycle with a clock frequency the same as the memory-module. The Altera Memory controller will be used in the design, this will take care of all the timing of the memory module, the controller can be run in either Half-rate mode where it sends 4*Data_bits in half the frequency of the memory-module and it can be run in full-rate mode where the controller will send 2*data_bits in the same frequency as the memory module.

There are two major methods of storing a dense matrix in memory, the first is row-major order where the matrix $\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ would be stored in memory row by row. a_{11} on address 0, a_{12} on address 1, a_{21} on address 2, a_{22} on address 3.

The other method of storing the matrix is column-major order where the matrix would be stored in memory column by column. a_{11} on address 0, a_{21} on address 1, a_{12} on address 2, a_{22} on address 3. To read the memory effectively

the best way of storing it is to store it so that the values that are needed after each other lies in a row-major way.

Access to the same row is fast, back to back read/write the read and write operations are pipelined, hence the CAS latency will be hidden. It costs time to change rows because they needs to be PRECHARGED and ACTIVATED. When the memory is accessed sequentially it will be most efficient since the PRECHARGE/ACTIVATE commands needs to only be performed every time the row is changed. It is also possible to have a row open in each bank. Hence if the data that are needed is known far ahead of time the row can be activated in the next bank while the last part of the current row is read.

Examples of overhead on the DDR-SDRAM data bus are:

- Activate time for new banks/rows
- Precharge time for changing rows within the same bank
- Write recovery time to change to read accesses
- Bus turnaround time to change from read to write
- Refresh time

Factors that effect the efficiency of the controller negatively includes:

- Individual reads
- Low burst-size
- Reading or writing to a closed row
- Refresh cycles
- Switching between read and writes

9.1 DDR1

Bandwidth between 1600 and 3200

Prefetch buffer depth is 2n bits

Voltage Supply : 2.5

9.2 DDR2

Bandwidth between 3200 and 8500

Prefetch buffer depth is 4n bits

Voltage Supply : 1.8 V

9.3 DDR3

Bandwidth between 6400 and 17000

Prefetch buffer depth is 8n bits

Voltage Supply : 1.5 V

Part III
Tools and programs used

Quartus II 10.0sp1

Quartus have been used to develop the VHDL code and test the ddr3 module and memory-controller. The TimeQuest analyser have been used to ensure that timing have been met. The signalTap Logic analyzer have been used to check if the right answer was found and the Virtual-JTAG was used to send data from the PC to the FPGA to be able to test the hardware. And program the FPGA through the JTAG cable.

Microsoft Visual Studios 10

Have been used to make random test matrices and write them to file so that they could be read by the VHDL-testbench. A program have been developed to convert the content in text files from hexadecimal to decimal and from decimal to hexadecimal. A software version of the Conjugate Gradient Method have been developed to verify the result that the FPGA came up with. This software version are also able to use Jacobi preconditioner.

Multisim-Altera 6.5e Starter Edition

Multisim have been used to simulate every module that was designed.

Arria II GX Development Kit

Has been used to download the design to the FPGA name? This have a ddr2 module, a ddr3 module and some leds and buttons. The Arria II GX was introduced in 2009. It was made for cost sensitive applications. It's logic fabric consist of adaptive look-up tables(ALUTs), registers, DSP blocks and embedded memory blocks. The Arria II development card from Altera is a midrange card, in performance it is between the low range card series Cyclone and the high range card series Stratix. It is a low-power and low cost card but is still able to support external memory with high bandwidths. DDR 3 with up to 400 MHz clock frequency and DDR 2 with up to 267 MHz.

Type	Quantity
Adaptive look-up-tables(ALUT)	99,280
Registers	99,280
Memory - bits	6,727,680
18 bit DSP blocks	576
Delay locked loops(DLL)	2
Phase Locked loops(PLL)	6

Table 11: Arria II Resources

Part IV
Architecture

The architecture of the design will be explained in this section.

10 Design goals

The main goal when doing the design has been to the f_{Max} as high as possible because to make the design efficient it is very important to get a high throughput. To be certain that the highest possible frequency will be reached, all the arithmetic modules have used the longest latency possible, 14 clock cycles for the addition and subtraction module, 11 clock cycles for the multiplier module and 24 clock cycles for the divider module. Another goal has been to keep as much of the variables in the Conjugate Gradient Method algorithm stored in internal memories to reduce the reading from external memory to the minimum. The only variable that can not be stored in the FPGA is the coefficient matrix A. To have a good control over the resources and f_{Max} of the different modules, the circuit has therefore been designed to be very modular. The control uses handshaking signals to communicate with the modules, matrix_vector multiplier and dot product, the control sends a start signal to the module when it needs that module to start its operation, and the modules sends a done signal back to the control when they are finished with the operation. The top module consist of the matrix-vector multiplier, the dot-product module, different arithmetic modules, on-chip memories to store the vectors in , register to store the scalars in and multiplexers to control the flow of data. The letter N will from now on mean the number of rows in the coefficient matrix A.

11 The memory module

All the vectors in the Conjugate Gradient Method will be stored in two-port RAMs in the FPGA logic fabric, this module have two different addresses, one for read and one for write, in addition, it have a read_enable and write_enable pin. The first element in the vector will be stored in address 0 and the last in

address N-1, the vector-elements will always be processed starting with the first element and ending with the last, hence it will be possible to simplify the control of the internal memories. A counter can be used on both the read_address and write_address port to generate the address, the read_address_counter and write_address_counter will be incremented by one by the read_enable and write_enable signal respectively. This has been done to simplify the reading and writing to the internal-memories, by doing it this way the address generation is performed automatically, hence the only operation that needs to be done to read and write to the memories is to put the read_enable port high for reading and write_enable_port high for writing. If any of the two counters reaches the last element that is stored in the memory it will roll around and start on address 0 again. This means that the memory module needs to know how many elements it is supposed to store, this is why the N signal is needed by the memory.The number of elements that are stored in the memory is N-1. The module can be seen in figure 5 .

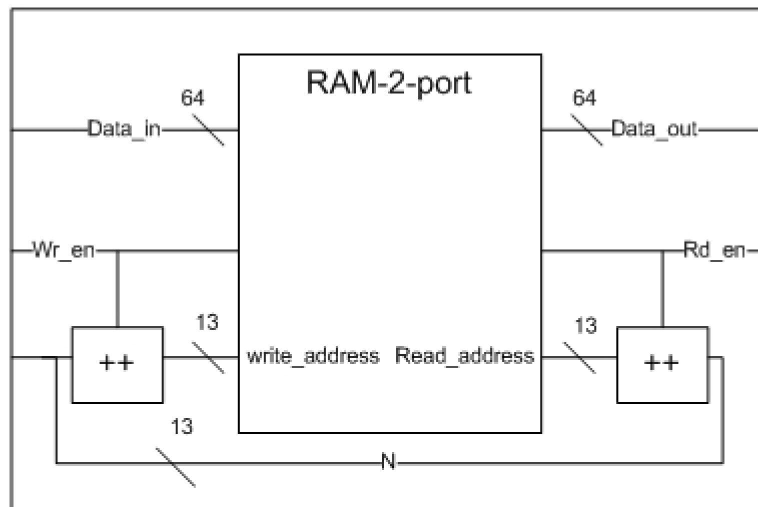

Figure 5: Memory Block

12 The Dot-Product Module

The dot-product module is responsible of performing the mathematical operation: $a \bullet b = \sum_{i=0}^{N-1} a_i b_i = a_1 b_1 + a_2 b_2 \dots a_{N-1} b_{N-1}$ where $a = [a_0, a_1 \dots a_{N-1}]^T$ and $b = [b_0, b_1 \dots b_{N-1}]^T$

In the conjugate Gradient algorithm it can be seen that the dot product operation is needed in two places. First it is needed to calculate the δ_{New} scalar by solving $r^T r$, secondly it is needed in the equation where α is solved to solve the inner product $d^T q$.

It is impossible that these two operations will need to be done at the same

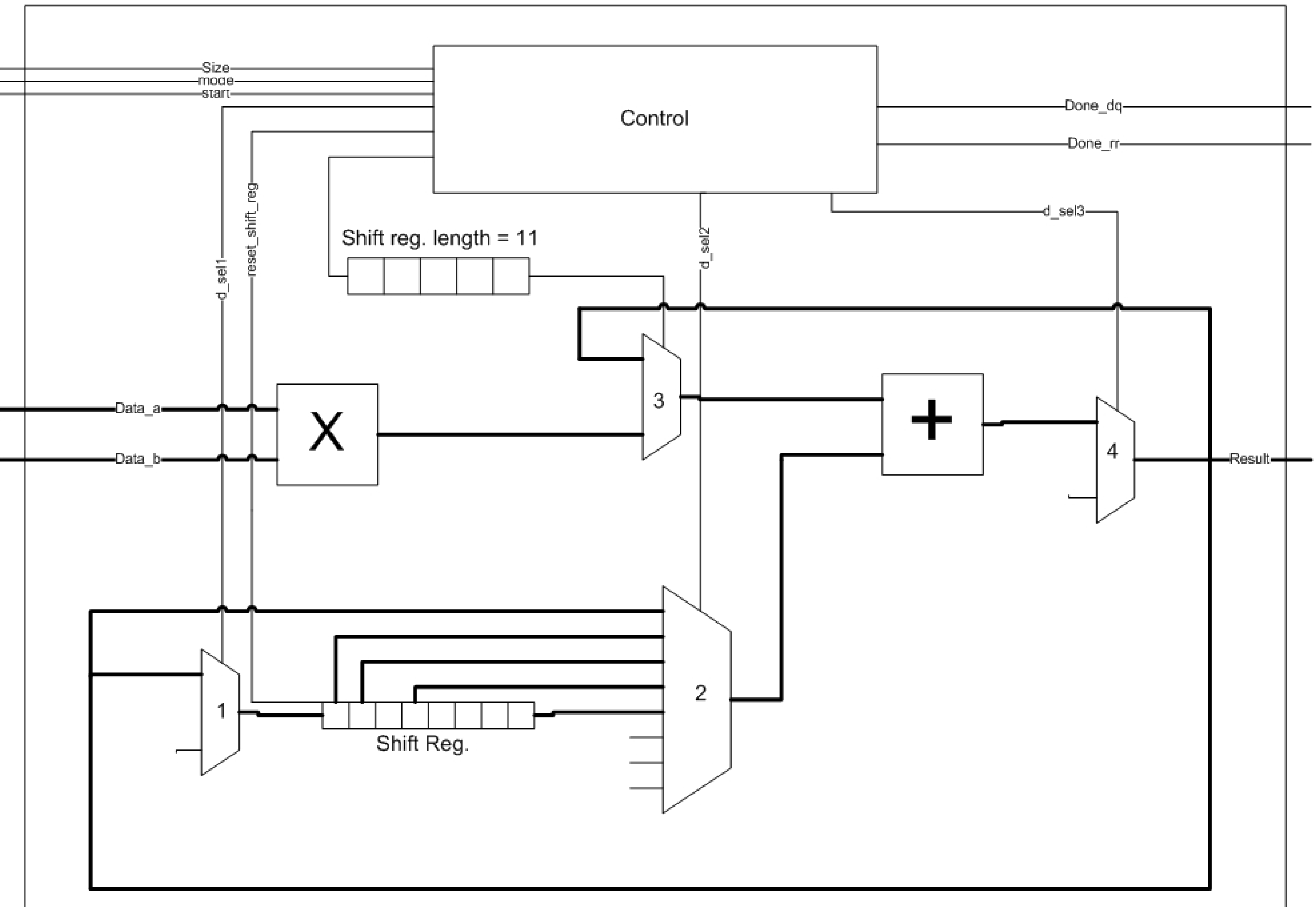

Figure 6: This is the dot product module

time, hence, only one dot product module is needed to be implemented into the circuit.

The circuit have two data inputs where the elements of the inputs vectors will come in and one data output, where the result of the dot product will be presented.

In addition, it have three control inputs, the N input is the number of elements in the vectors. To be able to differentiate between the two dot-products $r^T r$ and $d^T q$ a control bit input called mode is used. When the module is going to perform $r^T r$ the mode will be set to one, else it will be set to zero, this is controlled by the top control module in figure 16.

The module have two output signals called done_dq and done_rr. Done_dq is used as the select bit on the input multiplexer to the $d^T q$ register. And Done_rr is used as the select bit on the input multiplexers to the δ_{New} register, this can be seen if figure 15. When the mode is one and the result is ready, the done_rr port will become high and hence store the result in the δ_{New} register. When the mode is zero and the result is ready, the done_dq will become high and hence, the result will be stored in the $d^T q$ register. This can be seen more clearly in figure 15.

The dot product module assumes that the elements in the input-vectors will come to the inputs with no idle cycles between them. Since these vectors are stored in on-chip memory this will not become a problem since the on-chip memories are able to give one output every clock cycle, after an initial read latency of one to the first element. Number of floating operations needed in this operation is N multiplications and N addition. The resources used by this module can be seen in figure 12.

Type	Quantity
f_{Max}	180MHz
LUTs	2265
Registers	2730
Memory bits	216
Multipliers	10

Table 12: Resources used by the dot-product module

12.1 The dot product control module

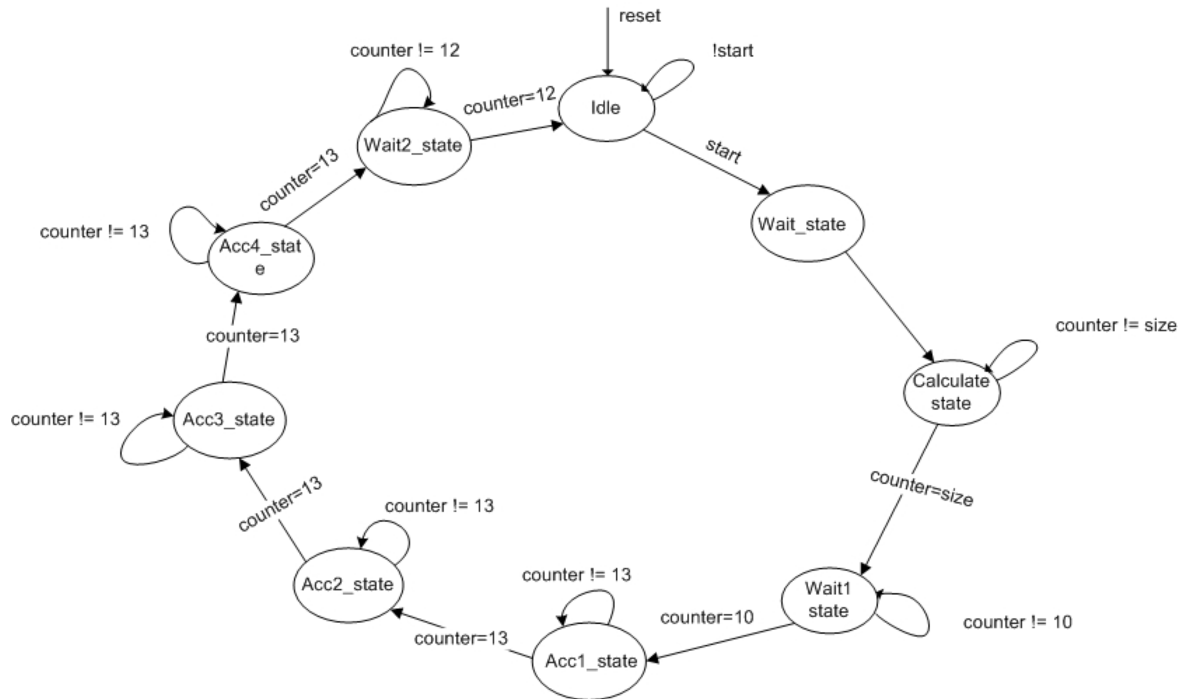

Figure 7: Dot-Product, State-machine

The control module of the dot product module consist of a state machine and a counter. When start goes high the state machine will go to the wait state. This state is needed because the start signal to the dot product module and the read signal to the memory block where the input vectors are stored is sent on the same clock cycle, and since the internal memory have a latency of one from the read_en signal goes high and to the data are presented at the output, the value will not be ready on the input port of the dot product before one clock cycles later, hence, the state machine needs to wait one clock cycle before it can start receiving vector elements. From the wait state it goes directly into the next state, which is the calculation state.

In the calculation state a pulse is sent to the shift register every clock cycle. When N pulses have been sent the state machine goes to the next state. The shift register is used to control the multiplexer that let values into the adder. Since the multiplier have a latency of 11 clock cycles the shift register needs to have a length of 11.

The state machine machine goes into a wait state to wait for all the data values to pass trough the multiplier and into the adder.

At this time 14 part sums of the total sum are stored in the adder pipeline.

All these sums needs to be added together to produce the final result. Hence a reduction circuit is needed. The reduction circuit is responsible of summing up all the values that are in the accumulators pipeline into one value. This can be best explained with an example. Imagine a four-stage adder-pipeline that are filled with the values depicted in figure 8. These values needs to be reduced to one value, hence the 1, 2, 3,4 needs to be added together to get 10. This is done by using another adder where one of the inputs is delayed by one clock cycle. This will result in the second pipeline shown in the figure after 4 clock cycles. In the next stage one of the inputs of the adder-pipeline is delayed by two clock cycles, after 4 clock cycles this will result in the values in the last adder-pipeline. The answer will then be ready on the end of the last adder after 3 more clock cycles, hence the total time the reduction will take is number_of_reduction_stages*pipeline_of_adder + pipeline_of_adder-1. The number of stages needed depends on the latency of the adder module used. If we called the latency of the adder module A_{Lat}, the number of reduction stages R_{Stage}, needed to reduce the values in the pipeline into one value could be calculated as $R_{Stage} = ceil(lg_2(A_{Lat)})$, Where the n-th reduction stage would have a delay of $R_{StageDelay} = 2^{N-1}$.

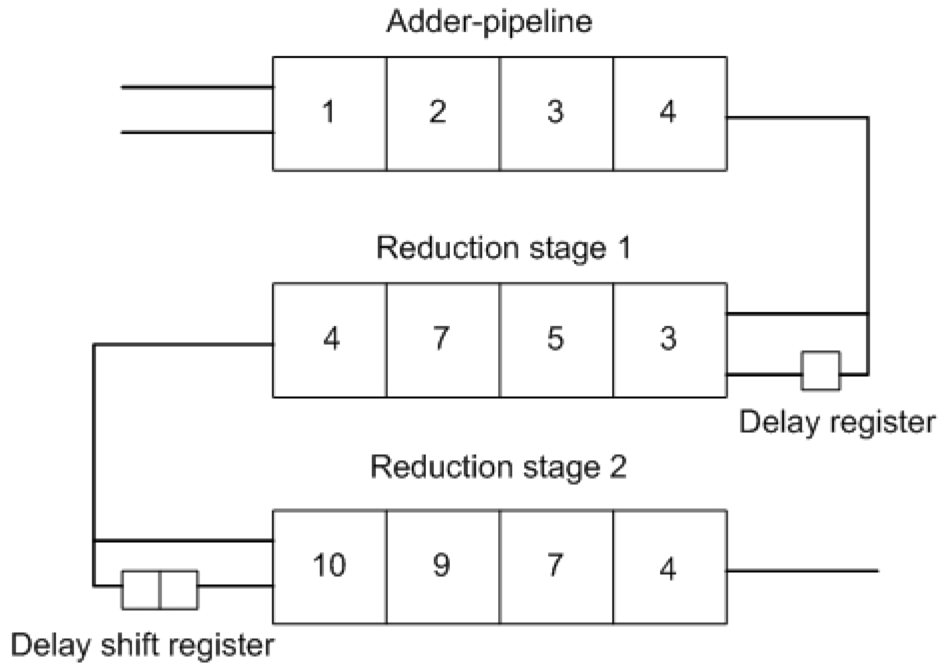

Figure 8: Reduction circuit example

Since the adder used in the dot-product have a latency of 14 clock cycles the number of reduction stages needed will be $R_{Stage} = ceil(lg_2(14) = 4$. Even though a new adder was used in every reduction stage in the example, it is

possible to use the same adder for all the stages and hence save resources. The delay is controlled by multiplexer 3 in figure 6. The multiplexer is controlled by the state-machine in figure 7. The state acc1_state is responsible of doing the first reduction, state_acc2 does the second reduction, state_acc3 does the third reduction and the acc4_state does the fourth reduction. In the wait_two state it waits until the final result is ready on the output. The control then sets either the rr_done or the dq_done high depending on the mode signal. If mode is one then rr_done goes high, else dq_done goes high.

13 The vector-update pipeline

In the Conjugate Gradient algorithm it can be seen that three of the operations that needs to be performed in every iteration are very similar, these can be seen in table ,

Similar operations
$x = x + \alpha d$
$r = r - \alpha q$
$d = r + \beta d$

Table 13: Similar operations

All of these operations can be done in a very similar way, how the circuit that does this operation look like can be seen in figure 9. To perform for instance the operation $x = x + \alpha d$, the multiplexer numbered 1, 2, 7, 9, 11, 12 and 14 needs to be on, all the multiplexers is controlled by the top_control module. Afterward, the control module puts the control_in signal high for N clock cycles, where N is the size of the vectors. Everything else will happen by itself as seen in figure 10. The signal will first give memory_d a read_enable signal and the first element of d will go into the multiplier and be multiplied with α, the control_in signal will then go into the shift register and give memory_x a read_enable signal after 11 clock cycles, this is exactly at the same time as the product αd is ready on the output of the multiplier module. Subsequently, this will make the first element in x go into the adder module and be added with the product αd who now comes out of the multiplier. Finally, After 27 clock cycles the control_in signal comes out of the shift register and gives a write_enable signal to the x_memory at exactly the same time as the sum $x + \alpha d$ is ready on the output of the adder module. The operation of the other two operations will happen in the same way, except that other multiplexers need to be on to be able to perform them. The alfa and beta signals comes from the divider circuit in figure 15.

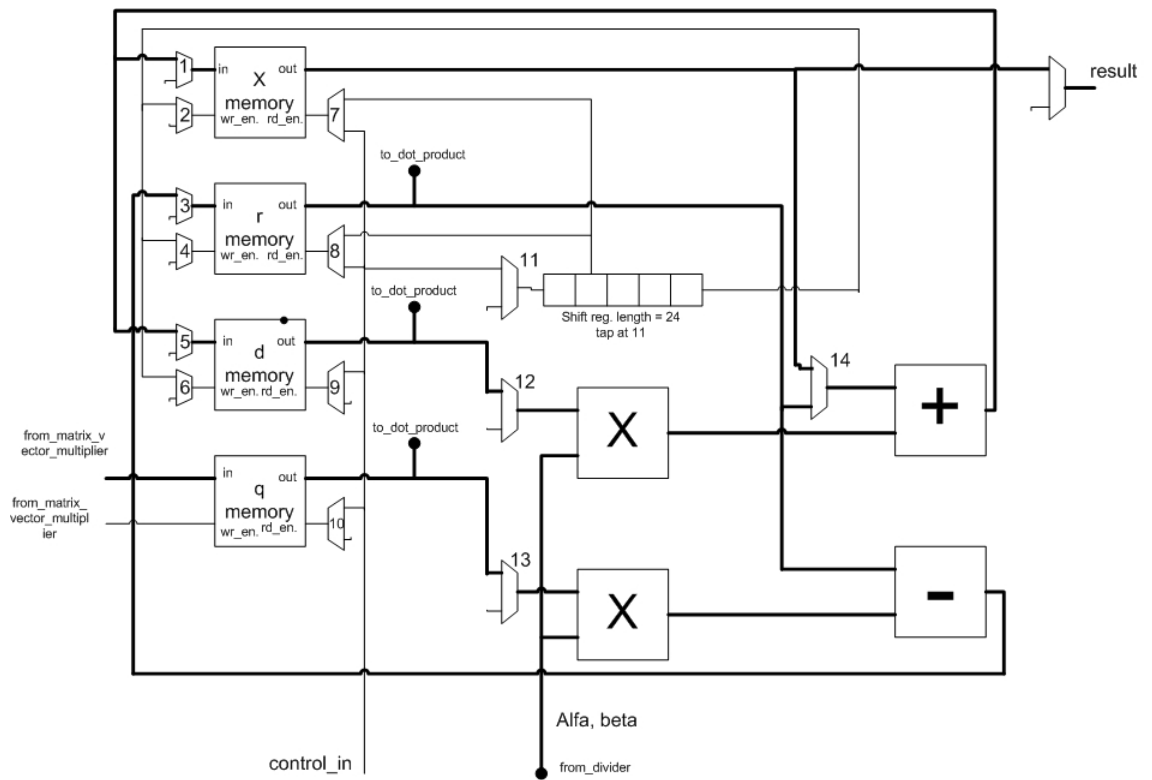

Figure 9: How the update of the vectors have been done

The pipeline of the memory module can be seen in figure 10.

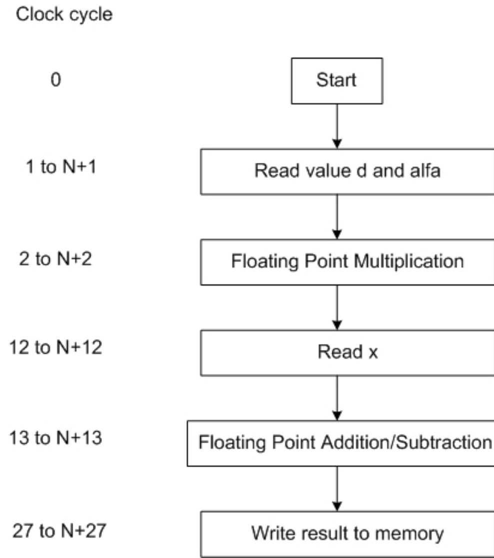

Figure 10: The Add Multiply Pipeline

14 Matrix-vector calculation

It can be seen in the Conjugate Gradient algorithm that a matrix-vector multiplication needs to be done in every iteration, hence a module is needed that are capable of doing the mathematical operation: $q_{(j)} = \sum_{i=0}^{N-1} A_{ji} d_i$, where A is an $N \times N$ matrix and d and q are vectors of size N. The module basically takes one row of A and takes the dot product of this row and d to produce the first value in q, it then continue with the next row in A and do the same N times. The number of operations required to do this operation is n^2 multiplications and n^2 additions. By using pipelining the additions and multiplications can be done at the same time after the pipeline has been filled up. By using parallelism it is possible to do more than one multiplication every clock cycle, hence this operation is very well suited for acceleration using an FPGA. One downside is that the matrix A will be to big to be able to be stored in the FPGA. Hence, it need to be stored in an external memory. The external memory is not as fast as the internal memory because of the bottleneck in the bus between the memory and the FPGA. This means that the speed of the matrix-vector multiplication will most probable be constrained by the external memory bandwidth.

By overlapping the supply of data with the processing of data a high throughput is possible. To make this possible the data needed by the matrix-vector multiplier, hence, the vector d and the elements in A that are next needed have to be readily available on the input pins. This have been made possible by storing every even numbered element in d on the input to the first multiplier and every odd numbered element on the input to the second multiplier. FIFOs

that act as temporary storage for the elements of the matrix that are read from the external memory have been put on the other input pin to the multiplier. Elements should always be buffered into the temporarily storage from the external memory, to ensure that the values will always be ready at the input of the matrix-vector multiplier module. By increasing the number of multipliers in the matrix-vector multiply unit it will be able to get a much higher throughput as long as the external memory bandwidth is able to supply the data needed. One multiplier is able to process one double precision floating point unit every clock cycle, which is a 64bit value. This means that the external memory need to have a bandwidth of at least $64M * CG_{freq} bit/s$ to be able to supply all the multipliers, where M is the numbers of multipliers and CG_{freq} is the clock frequency of the module. If M is 2 the required bandwidth is 2.24GB/s assuming a clock frequency of 150MHz.

The memory usually have a long latency but when this is over it is able to give out one value every clock cycle. It is very important to utilize the memory throughput with maximum efficiency.The external memory bandwidth to the FPGA will usually be ... by the number of pins on the device. There are FPGAs on the market today that are capable to interface with seven external ddr3 memory modules and hence get a bandwidth of 70 GB/s. The number of bits the matrix-vector multiplier needs every clock cycle depends on the amount of parallelism used, for every multiplier it needs 64 bits. The external memory usually runs on a higher clock frequency and is therefore able to give more than one for every value the multiplier processes. It is therefore important to use FIFOs as buffers between the external memory controller and the matrix-vector multiplier. The way the matrix vector multiplier needs the values that are stored in ram is very important, because the extremal memory have a longer latency changing bank and row and it is therefore important to read all the values in the column before going to the next row. It is very efficient to read the memory from the top left to the bottom right. Therefore the values needs to be stored in the memory in the order that they are needed by the matrix-vector multiplier to be able to take full advantages of the available external memory bandwidth.

The matrix-vector multiplier consist of four main parts.

The adder-tree, if both of the inputs have a value these will be added to each other, if only one of the inputs have a value and the other have a zero, the value be added to zero, hence the values that are ready on the inputs will always come trough, it does not need to wait until all of the inputs have a value ready to do the operation. All the values will only be added to the final result once. The adder tree can be seen in figure 11.

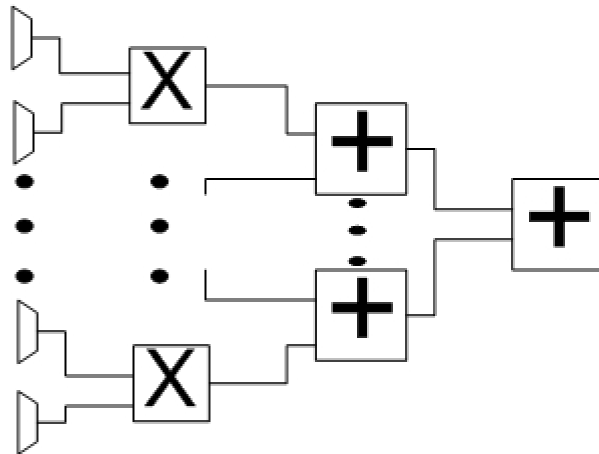

Figure 11: The adder tree

The accumulator accumulates all the results. This is needed because the multipliers will not be able to multiply all the inputs in one row of the matrix in one clock cycle if the matrix is big, because of too few resource on the FPGA and because the matrix is stored in external memory and the external-memory-bandwidth will not be able to supply all the values from one row in one clock cycle. The accumulator sums all the values in one row when the row has been sent trough, the accumulator will have part-values of the row spread in its pipeline these needs to be reduced to one value before it is stored in q_memory. The accumulator can be seen in figure 12.

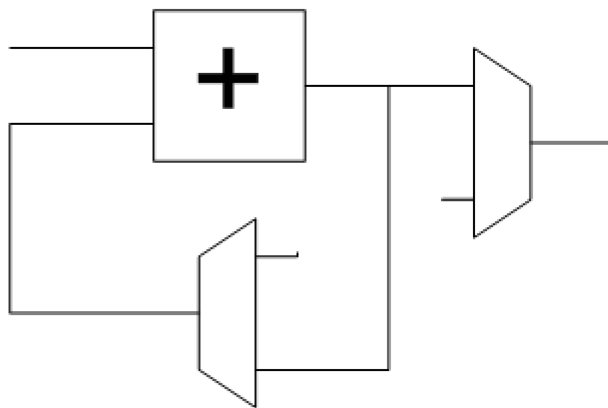

Figure 12: The Accumulator

The reduction circuit is responsible for summing up all the values that are in the accumulators pipeline into one value. The matrix-vector multiply module

will, in contrast, to the dot product get new values in after the first dot-product have been produced. Naturally, this means that the accumulator can not function as the reduction circuit as it did in the dot-product, that would create a big pipeline stall. To avoid a pipeline stall, the reduction circuit have been rolled out as can be seen in figure 13. This will use much more resources compared to the solution used in the dot-product, but it is much more important to get a high throughput than it is to get an area-efficient design.

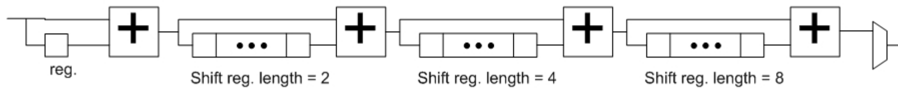

Figure 13: Reduction Circuit

The control module consist of one state-machine for each multiplier, this has been done so that all of the multipliers can work independently of each other, they do not need to wait for each other. These state-machines reads in their part of the row and when finished goes into a wait-state. A synchronizing state-machine have been used to synchronize all the input-state-machines. It waits for all the input-state-machines to go into wait state. It then sends out signals to the accumulator to release the values it have in its pipeline and sends them through the reduction circuit, it also sends out row_ done signal and done signal if all the rows have been processed. Since all these values are sent out on the same time they need to be delayed by shift registers to ensure that they arrive at the destinations at the right time as they are needed.

If the matrix is over 14×14 this module will be able to pipeline all the operations without any stalls in the pipeline as long as high external bandwidth is assumed. If it is less than 14×14 the pipeline needs to stall when the accumulators pipeline is sent to the reduction circuit to ensure that the elements from the next row is not mixed with the elements from the last row in the reduction circuit.

Resources used by the matrix-vector module can be seen in table 14.

Type	Quantity
f_{Max}	195 MHz
LUTs	9700
Registers	9830
Memory bits	2192
Multipliers	20

Table 14: Resources used by the Matrix vector multiplier

The complete matrix-vector circuit can be seen in figure 14.

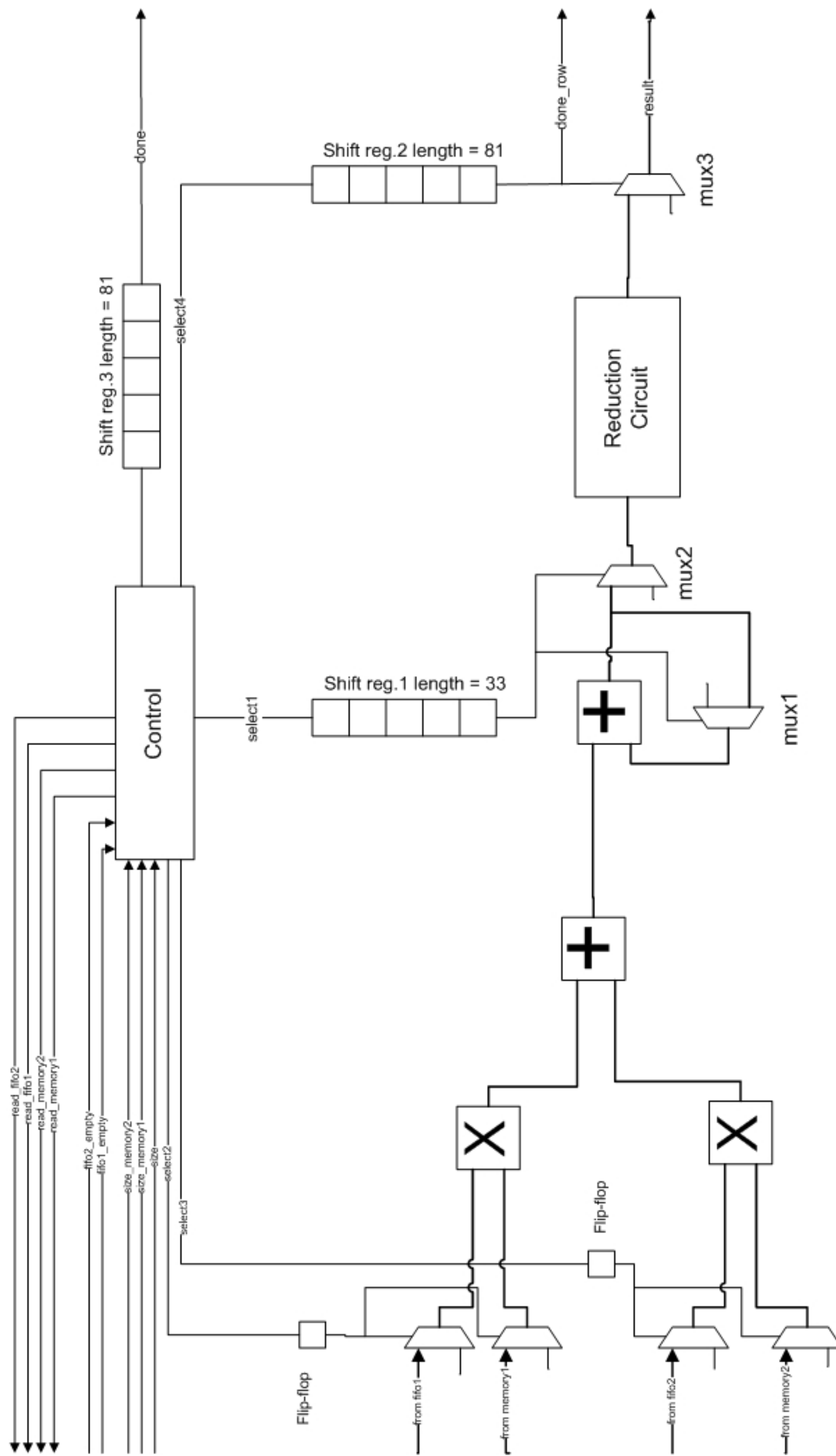

Figure 14: The Matrix-vector multiplier

15 The Top module

The divider can be seen in figure 15. In the upper part of figure 15 the circuit responsible for the stop condition can be seen $\| r_{(i)} \| < \varepsilon^2 \| r_{(0)} \|$. The ε^2 are stored in the register in the initialization part of the algorithm. This is multiplied by the $First\delta_{New}$ and the result is compared to the δ_{New}. If δ_{New} is smaller, the algorithm will stop and the answer has been found. If not it will continue with a new iteration. In the lower part of figure 15 the registers in the algorithm can be seen in addition to the divider. When the done_rr port from the dot product module goes high the result from the dot product will be stored in δ_{New}. If the done_dq port from the dot product goes high the result will be stored in the $d^T q$ register. When multiplexer 5 goes high the value in register δ_{New} will be transferred to register δ_{Old}. even though there are two divisions in the Conjugate Gradient Module only one divider needs to be implemented, because the two divisions will never happen at the same time. When a division is needed to be performed the control module opens up multiplexer 6 to let δ_{New} trough and opens up one of the ports in multiplexer 7 depending on if δ_{Old} or $d^T q$ is going to be divided by. The default value out of multiplexer 7 is 1 to avoid dividing by zero. In this circuit it can also be seen very clearly that if the value in one of the registers δ_{Old} or $d^T q$ is zero, a divide by zero will happen if the algorithm has not stopped. This is something that have to be avoided. A divide operation take 24 clock cycles, hence, the control signal needs to be delayed by a shift register until the divide operation is finished and a result is ready on the output, the control signal will then open up multiplexer 8 and let the result of the division into the α, β register. α and β can share a register because the two signals will never be needed at the same time, this can be seen in the Conjugate Gradient algorithm. When the α or β are needed the multiplexer 9 will open and send them to the lower input of the multipliers in figure 9.

Top module control

The top control module is responsible for controlling the operations trough the Conjugate Gradient method, a figure of the state machine can be seen in 16. It sends out start signals to the different modules in the circuit and get done signals back from the modules when they are finished, in addition, it also control the multiplexer in the circuit. A counter in the state machine controls the time sequence in every state, this counter will be reset when the state-machine changes state, and therefore can be reused in the next state. Here follows a description of the state machine.

In IDLE it will wait for the start button to be pushed, when pushed it will go to INIT_STATE.

In INIT_STATE the error tolerance is transferred to the error register this can be seen in figure 15. The known vector b is transferred to d-memory and r-memory. x-memory is filled with zeros. The calculation of the size of the memory modules in front of the matrix-vector multiplier is also done in this

Figure 15: Divider Circuit

state. If done_size = 1, the state machine will go to the next state, calculate_rr_init_state. It will use N clock cycles in this state.

In CALCULATE_rr_INIT_STATE the dot product $r^T r$ will be calculated, by sending a start signal and the r values to the dot product module. When the done_rr signal goes high the result is transferred to the δ_{New} register, this can be seen in figure 15, the next state it goes to is PUT_D_NEW_IN_D_NEW_FIRST_STATE.

In PUT_D_NEW_IN_D_NEW_FIRST_STATE the δ_{New} is transferred from δ_{New}-register to $\delta_{Newfirst}$-register by opening up the multiplexer between them. The next state is FILL_MEMORY_STATE.

In FILL_MEMORY_STATE it sends a start signal to the fill-memory-module and sends the d-values to it. When done_fill_memory=1 it will go to the next state which is the MATRIX_VECTOR_STATE

When in MATRIX_VECTOR_STATE it sends a start signal to the matrix-vector-module. Then it waits for the done signal from the matrix-vector module. When the done_matrix_vector signal goes high it goes to the next state which is the CALCULATE_dq_STATE, number of clock cycles used here depends on the number of clock cycles the matrix-vector-module uses, this again depends on how many clock cycles it take to read the matrix A from the external memory.

In the CALCULATE_dq_STATE it sends a start signal to the dot product module and sends the d and q from their memories and to the dot product, the mode of the dot-product is set to zero. When the signal done_dq=1 the next state will be the CALCULATE_ALFA_STATE.

In CALCULATE_ALFA_STATE the δ_{New} and $d^T q$ is sent to the divider, if done_division=1 the next state will be x_AND_r_state.

In x_AND_r_state the x and r values is updated in parallel. Next state is PUT_DELTA_NEW_IN_DELTA_OLD.

In PUT_DELTA_NEW_IN_DELTA_OLD the δ_{New} will be transferred to the δ_{Old} register. Next state is CALCULATE_rr_state.

In CALCULATE_rr_STATE a start signal is sent to the dot-product-module and mode=1, and r will be transferred from memory to the dot-product-module. Next state is STOP_OR_NOT_STATE.

In STOP_OR_NOT_STATE the operation $\delta_{New} < \epsilon^2 \mid b \mid$ will be performed, if true next state is SEND_OUT_X_STATE else next state is CALCULATE_BETA_STATE.

In the SEND_OUT_x_STATE the x values will be sent from x-memory to the result FIFO. Next state is Idle.

In CALCULATE_BETA_STATE the δ_{New} and δ_{Old} is sent to the divider to calculate β. If done_division=1, next state is CALCULATE_d_STATE.

In CALCULATE_d_STATE the d will be updated, and next state is FILL_MEMORY_STATE.

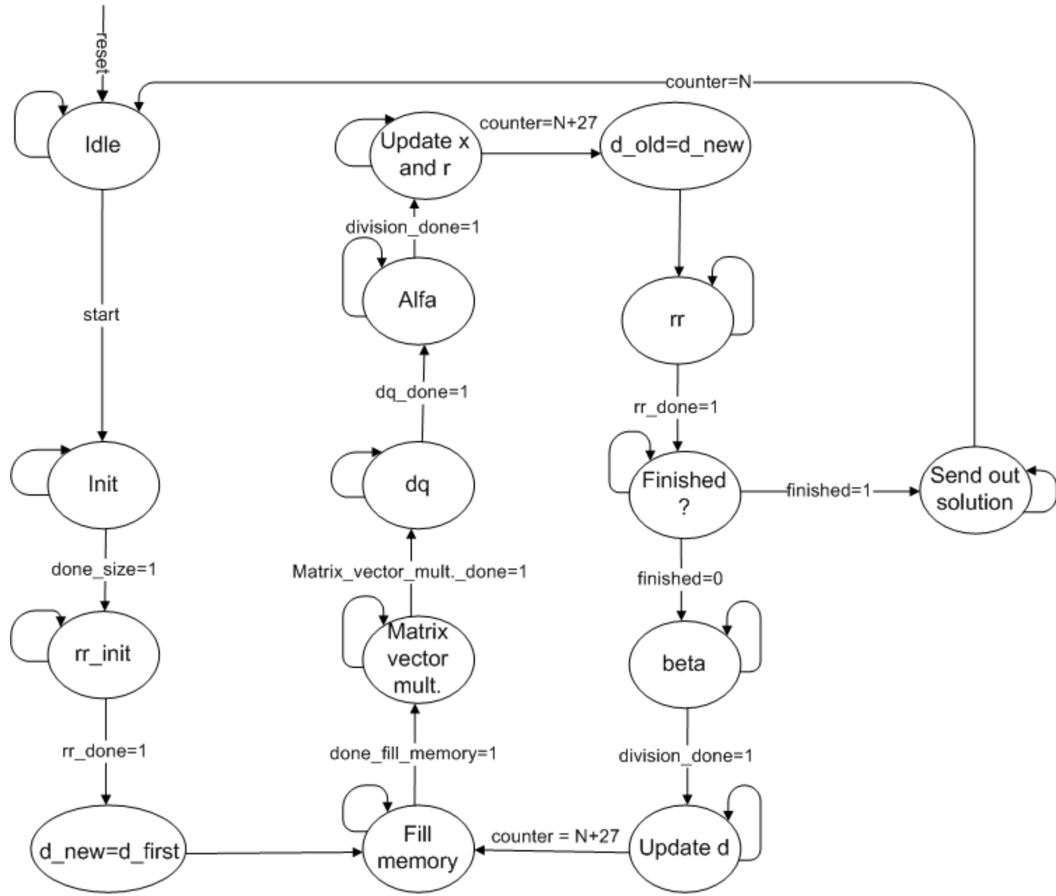

Figure 16: Top module control

Table 15 describes what the different variables used in table 16 means.

Variable	Description
N	Number of rows in A
M	Number of multipliers in the matrix-vector multiplication module
A_{Lat}	Latency of floating point addition
M_{Lat}	Latency of floating point multiplication
D_{Lat}	Latency of floating point division

Table 15: Variable names

The number of clock cycles used in the different states in the control module can be seen in 16. As can be seen the matrix-vector multiplication $q = Ad$ is dominating the total number of clock cycles as was predicted. Hence, even though there are some small optimizations that can still be done it will not have

much effect on the total run time. The latencies can also be reduced and still get a high clock frequency, but the improvement it will have on the total run time will be negligible. The only improvement that can be done and have a high effect on the run-time is to use more multipliers in the matrix-vector multiplier that works in parallel. In table 16 only two multipliers have been used. However, when the number of multipliers are increased the external memory bandwidth needed to be able to supply the multipliers with data will also be increased.

Operation	Clock Cycles/iteration	Clock cycles in this design with N=8192 and M=2
Calculate size	$N+1$	8193
$\delta_{New} = r^T r_{Init}$	$5A_{Lat} + M_{Lat} + N + 1$	8274
$\delta_{NewFirst} = \delta_{New}$	1	1
Fill Memory with d	$N+2$	8194
$q = Ad$	$A_{Lat} lg_2(M) + 5A_{Lat} + M_{Lat} + 1 + \frac{N}{M} + (N-1)(A_{Lat} + \frac{N}{M} + 1)$	4192+33673201
$d^T q$	$5A_{Lat} + M_{Lat} + N + 1$	8274
α	D_{Lat}	24
Update x and r	$N + A_{Lat} + M_{Lat} + 3$	8220
$\delta_{Old} = \delta_{New}$	1	1
$\delta_{New} = r^T r$	$5A_{Lat} + M_{Lat} + N + 1$	8274
$\delta_{New} < \varepsilon^2 \delta_{NewFirst}$	M_{Lat}	11
β	D_{Lat}	24
Update d	$N + A_{Lat} + M_{Lat} + 3$	8220
Total clock cycles/iteration		33735103

Table 16: Theoretical iteration time

Part V

Testing and verification

The testing and verification of the Conjugate Gradient method has proved to be very challenging because of the big size of the matrix. The matrix needed first to be made and then to be transferred from the host-PC to the external memory on the Arria II development card. Hence, a complete test environment had to be developed. To be able to generate symmetric positive definite matrices a matrix-generator program was made in Visual Studios, this program generates a file with a symmetric positive definite matrix, the b vector and the size of the matrix. This needs to be converted from decimal to hexadecimal, hence a program that does this have been developed in Visual Studios. The syntax of the file containing the size, b-vector and matrix A can be seen in figure 17.

Hexadecimal value	Decimal value	Line number in file	Variable
0000000000000002	2	1	N
4008000000000000	3	2	$b_{(1)}$
4010000000000000	4	3	$b_{(2)}$
4008000000000000	3	4	$A_{(1)(1)}$
4000000000000000	2	5	$A_{(1)(2)}$
4000000000000000	2	6	$A_{(2)(1)}$
4010000000000000	4	7	$A_{(2)(2)}$

Table 17: File format

Virtual JTAG is an Altera IP that uses the byteblaster download cable to make a communication channel between the host-PC and the altera development-board, this has been used to transfer the content of the file to the FPGA. A tcl-script have been made that is able to control the Virtual JTAG. The Tcl-script contains two functions, PUSH and POP.

When PUSH is used, the content of the file is read and converted to bits, the bits are then sent trough the USB-cable to the FPGA serially bit by bit, when 64-bit has been sent trough, the 64-bit value is stored in a receiver-FIFO on the FPGA the test system can be seen in figure 18.

The test state machine is then reading the content from the FIFO and sends the size and b to the Conjugate Gradient module and the matrix is sent to the ddr3 memory module on the development kit. The state diagram of this control module can be seen in figure 17.

A short description of what is happening in every state can be seen here:

In IDLE it waits until the external memory have been initialized, then it goes to size_state,

In SIZE_STATE it reads first value in jtag_fifo, this is the size of the matrix, as can be seen in figure 17.

In STORE_SIZE it stores the size in the size register on the FPGA

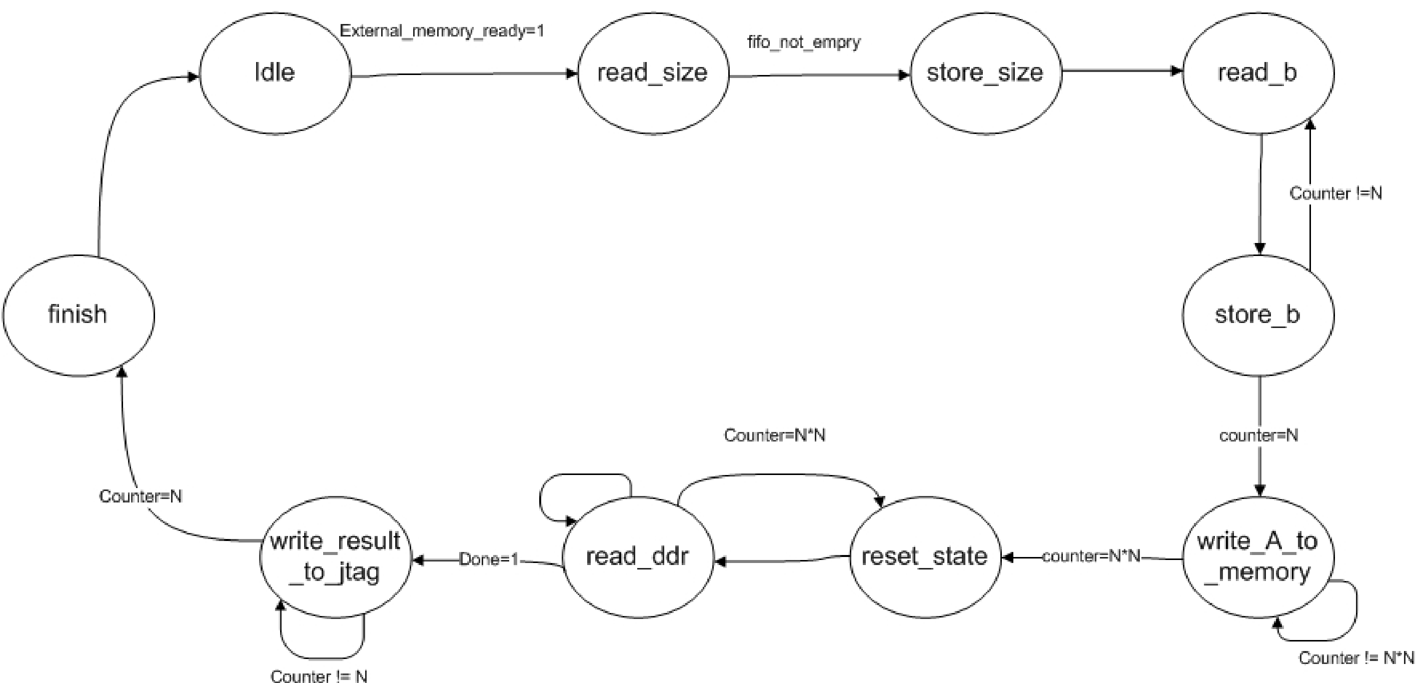

Figure 17: Test control machine

In READ_REQ_TO_B it reads the next values from the jtag fifo

In WRITE_TO_B it write the b values to the conjugate gradient method

In WRITE_TO_DDR it write the matrix from the JTAG-fifo to the external memory on the development card

In RESET_STATE it reset the ddr_address

In READ_DDR it reads the DDR3 and sends the content to the fifos in the Conjugate Gradient method, it will be in this state until the system of linear equations is solved, it then goes to wait state.

In WAIT_STATE it wait until the solutions are transferred from the x_memory in the Conjugate Gradient method to the result fifo.

In WRITE_RESULT_TO_JTAG it reads the content of the result fifo and writes it to the JTAG_FIFO

When FINISHED STATE it goes to IDLE

When the result has been transferred to the FIFO, the POP function in the Tcl-script can be used to send the solution from the FIFO to the PC, it is then saved in a file on the PC.

A software version of Conjugate Gradient method have been developed so that the result coming out from the FPGA can be compared with the solution the software version of the Conjugate Gradient Method finds. The test system can be seen in figure 18.

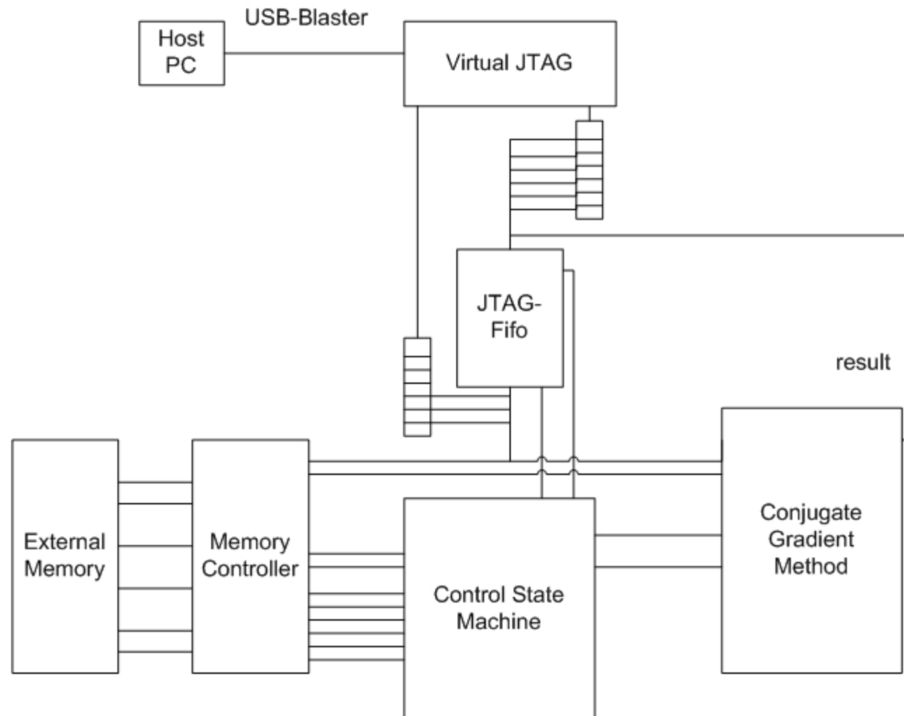

Figure 18: Test System

Because the JTAG can only send and receive one bit each clock cycle it needs to send the bits into a serial in parallel out shift register before sending the value in the shift register into a temporary storage FIFO.

Altera offers much verification tool that has been very helpful in debugging and testing the circuit. SignalTapII has been used to see the values on different nodes in the circuit in run-time. SignalTapII is the logic analyzer offered by Altera it is connected to the JTAG connection and sends the data from the nodes in the FPGA that is specified and sends them to the PC through the byteblaster-cable when a trigger condition happens. The trigger is specified by the designer.

SignalTapII is an embedded Logic Analyzer that can be programmed to trigger when a signal goes to a certain state, the internal signals on the FPGA can then be seen.

This file can then be read and compared with the solution found using the software version of the conjugate gradient method to see if they found the same result.

The ArriaII have two push-buttons that can be used by the user, these buttons have been used as reset and start buttons. PB0=reset, PB1=start.

Part VI
Results

16 External memory bandwidth needed

Since the only module that reads from memory are the matrix-vector multiplier, the external memory bandwidth needed are only dependent on how many multipliers that work in parallel in this module. Since every multiplier in the module needs one new 64bit value from the external memory every clock cycle the memory bandwidth needed is $Bandwidth_{Min} = 64Mf_{CG}bit/s$, where M is the number of multipliers in the matrix-vector multiplier and f_{CG} is the clock frequency of the Conjugate Gradient Module. A figure taken from SignalTapII shows the memory-efficiency of the module, this can be seen in figure 19 . A new value from the external memory is available every time the local_rdata_valid signal is high, as can be seen in the figure this signal is high almost all the time, hence the memory efficiency can be seen to be over 90 %.

The module have been tested with different sized internal memories to see if it effected the maximum frequency, the results can be seen in 18. The max frequency where not much effected by routing congestion.

f_{Max}	Memory bits used	N
171	2070663	2048
163	2857095	4048
166	4429959	8096

Table 18: Max size effect frequency

The total resources used by the module can be seen in table 19. As can be seen there are still much resources that can be used to implement more multipliers that can work in parallel in the Matrix-vector module. However to get an effect out of the extra multipliers, an higher memory bandwidth is needed. Most FPGA have many I/O pins that can be used to connect many external memory modules.

Type	Quantity	% of total
LUTs	23506	24%
Registers	20956	21%
Memory-bits	3469127	51%
Multipliers	104	18%
PLL	1	17%
DLL	1	50%

Table 19: Total resource usage by the design

Figure 19: Memory bandwidth efficiency

The software version of the Conjugate Gradient method and the version implemented on the FPGA have been tested and compared, the result can be seen in table 20. It can clearly be seen that the FPGA are capable of performing much better than the CPU, even though both the CPU and the FPGA use the same amount of iterations the CPU use much more time per iteration. As the matrix size grows the difference between the FPGA and CPU grows, this is believed to be be because the larger the matrix the more the CPU needs to read from main memory, since it can not fit the entire matrix in its cache when the matrix is large. It have to be noted though that it is probably possible to make a much more optimized software version compared to the version that have been used in this testing. But since no other optimized software version was available a custom version had to be made. The specifications on the PC that the software version was run on can be seen in appendix A.

To test the effect of the Jacobi preconditioner, the software version was implemented with the Jacobi preconditioner. The matrices used in the test have been diagonally dominant, hence the Jacobi preconditioner should be very effective. It can be seen in 20 that it where able to reduce the number of iterations needed by 20-25%. However, the time used by the CPU to solve the problem increased. This is believed to have happened because of the extra matrix-vector multiplication needed in every iteration when the preconditioner is used. The matrix-generator program used to generate the test matrices were not able to generate positive definite matrices over 1600 and that is why bigger matrices have not been tested. But if we extrapolate the result it seems that the time the CPU use to solve the problem is doubled when the matrix is increased by 100. The FPGA does not have enough data to be able to extrapolate it, but the data in table 16 can be used to calculate the theoretical time that will be used by the FPGA.

Size of the matrix	λ_{Min}	λ_{Max}	κ	$CPU_{Time}[s]$	$CPU_{Iterations}$	$FPGA_{time}[s]$	$FPGA_{Iterations}$	CPU_{Time} With Jacobi preconditioner	$CPU_{Iterations}$ With Jacobi iterations	Improvement in iterations with Jacobi preconditioner
200	413	2315	5.6	0	8	0	9	0	6	25%
300	364	3167	8.7	0	9	0	11	1	7	22%
400	324	4024	12.4	2	10	0	11	2	8	20%
500	290	4870	16.8	3	12	0	14	4	9	25%
600	260	5714	22	3	13	0	10	6	11	15.3%
700	229	6572	27.8	4	14	0	15	8	11	21.4%
800	203	7407	36.5	8	16	0	18	10	12	25%
900	176	8272	47	10	17	0	19	14	13	23.5%
1000	149	9112	61.2	14	19	0	18	19	15	21%
1200	105	10815	103	25	23	0	22	30	17	26%
1400	65	12539	192.9	43	31	1	28	53	23	26%
1600	30	14216	478.6	80	46	1	43	102	35	24%

Table 20: CPU and FPGA run-time comparison

Part VII
Conclusion

In this paper the possibility of accelerating the Conjugate Gradient Method for dense matrices by implementing it on an FPGA have been investigated. It has been shown that the only variable in the whole algorithm that needs to be stored in external memory is the coefficient matrix A, since no writing to the external memory is needed, and since the matrix can be stored in the external memory in the order that it is needed it will be much easier to use the available external memory bandwidth with full efficiency. No round off error have been detected, but the matrices used to test the module have been well-conditioned, hence round off error may have an effect if the matrix is ill-conditioned and therefore more sensitive to round off error. The largest matrix-size that can be solved by this design will be constrained by the FPGAs internal memory, because all the vectors are stored in internal ram. The Arria II will be able to solve matrices up to 8192x8192. The FPGA implementation has been compared to a software implementation, it has been shown that the Conjugate Gradient Method are able to perform 35 times better than a CPU implementation of the Conjugate Gradient algorithm.

A big advantage of this architecture is that the matrix can be read constantly in the background as long as the FIFO buffers on the FPGA are not full. The FPGA are able to utilize 90 % of the full memory bandwidth capacity available, since only the matrix needs to be stored in external memory, all the memory bandwidth can be used for data transfers. The module has been able to get a relatively high clock frequency. It was possible to get over 160 MHz. The module that constrained the maximum frequency was as predicted the floating point division module.

17 Further work

It was showed that the Jacobi preconditioner were able to decrease the number of iterations needed to converge to the solution by 20-30% in the software version. It was not enough time to implement this on the FPGA, hence this is something that can be done in a later work. The exact solution to the residual $r = b - Ax$ can be implemented, and hence get rid off accumulated round of error in the residual. Larger and more ill-conditioned matrices needs to be tested to see how the round off error developes.

References

[1] R. Barrett. *Templates for the solution of linear systems: building blocks for iterative methods.* Number 43. Society for Industrial Mathematics, 1994.

[2] P. Belanović and M. Leeser. A library of parameterized floating-point modules and their use. *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, pages 657–666, 2002.

[3] G. Govindu, R. Scrofano, and V.K. Prasanna. A library of parameterizable floating-point cores for fpgas and their application to scientific computing. In *Proceedings of the International Conference on Engineering Reconfigurable Systems and Algorithms.* Citeseer, 2005.

[4] G. Govindu, L. Zhuo, S. Choi, and V. Prasanna. Analysis of high-performance floating-point arithmetic on fpgas. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 149. IEEE, 2004.

[5] Torstein Habbestad. A feasibility study of solving large dense systems of linear equations by using the conjugate gradient method implemented on an fpga, December 2010.

[6] M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal cf Research of the National Bureau of Standards Vol*, 49(6), 1952.

[7] N.J. Higham. The accuracy of floating point summation. *SIAM Journal on Scientific Computing*, 14:783–783, 1993.

[8] J. Liang, R. Tessier, and O. Mencer. Floating point unit generation and evaluation for fpgas. In *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, pages 185–194. IEEE, 2003.

[9] C.J. Lin and R. Saigal. An incomplete cholesky factorization for dense symmetric positive definite matrices. *BIT Numerical Mathematics*, 40(3):536–558, 2000.

[10] A. Lopes and G. Constantinides. A high throughput fpga-based floating point conjugate gradient implementation. *Reconfigurable Computing: Architectures, Tools and Applications*, pages 75–86, 2008.

[11] G.R. Morris and V.K. Prasanna. An fpga-based floating-point jacobi iterative solver. 2005.

[12] F. París and J. Cañas. *Boundary element method: fundamentals and applications.* Oxford University Press, 1997.

[13] E. Roesler and B. Nelson. Novel optimizations for hardware floating-point units in a modern fpga architecture. *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, pages 323–345, 2002.

[14] J.R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[15] K.D. Underwood and K.S. Hemmert. Closing the gap: Cpu and fpga trends in sustainable floating-point blas performance. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 219–228. IEEE, 2004.

[16] B. ZHANG, G. GU, L. SUN, and Y. WU. Floating-point fpga gaussian elimination in reconfigurable computing system. *Chinese Journal of Electronics*, 20(1), 2011.

Part VIII

Appendix A

A PC-Specifications

- Multicom Compal PBL21 15.6"
- OS : Windows 7 Home Premium
- CPU : Intel Core i7-2720QM four core, 2.2 GHz, 6MB SmartCache
- Memory: 8 GB DDR3 1333MHz (2×4 GB)
- SSD : Intel X25-M 120 GB