NTNU
Norwegian University of
Science and Technology

Implementing Controller Strategies in FPGA

Erik Normann Næss

Master of Science in Engineering Cybernetics
Submission date: June 2009
Supervisor: Sverre Hendseth, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem Description

Some controllers have to be very fast, and one solution is to implement them in FPGA-circuits.

This assignment is to create a library of blocks for some controller types, namely PID and MPC (Model Predictive Control). Central to the implementation would be to achieve greatest possible performance while using the least possible amount of resources on the FPGA.

By implementing the controllers in FPGA, it should be possible to implement different realizations of arithmetic operators, which should be considered for increasing the performance.

Implementations on FPGAs also give the possibility to implement controllers in parallel, which should be reflected by the implementations of the controllers in the library.

The controller parameters should be configurable, for instance by using a serial interface.

Assignment given: 12. January 2009
Supervisor: Sverre Hendseth, ITK

Abstract

In todays industrial applications there is an increasing demand for good control algorithms and implementions. This may be because of increased competition leading to smaller economic margins, safety reasons or even environmental reasons.

With such demands comes the need for faster and more efficient hardware. Unfortunately, even though CPU-speed has more or less sky-rocketed the last decades, using such solutions for a typical embedded system is not very cost efficient, practical or robust. Thus, many embedded systems these days use complete microcontrollers, such as the ATMEL AVR-chips. These however, run on far slower clock speeds than pure CPUs, and are not capable of performing the calculations needed for real-time controlling using for example an MPC controller.

One way of getting around the performance issue, could be to construct the controller entirely in hardware, designed specifically for the task at hand. This paper will look at how this is possible to accomplish by using an FPGA, and how much performance gain it is possible to achieve on this platform.

Preface

This master thesis is the result of 5 months of endless synthesizing of FPGA code on behalf of Siemens Oil & Gas in Trondheim. The work took place during the spring semester of 2009.

As my previous knowledge on FPGAs was basically non-existing, this has been a great opportunity for me to learn about a new and exciting field. In the beginning of the project a lot of time was spent trying to learn VHDL. This meant working my way up from blinking LEDs up to a complete VGA driver showing graphs for the implemented PID-controller. It’s amazing how much one can do with an FPGA!

Of course I have learned about more than just FPGAs from these months. MPC and QP-solvers are fields one can never have read too much about, and actually implementing and analyzing this gives a greater understanding on how they work in general.

I would like to thank Karstein Kristiansen at Siemens and Sverre Hendseth at NTNU for giving me the opportunity to work with this project, and lending me the Xilinx FPGA development kit. I would also like to thank Stefano Bertelli at NTNU for supplying me with the NI-DAQ board.

Table of Contents

1 Introduction

The purpose of this paper is to shed some more light on the use of FPGAs to implement advanced controller strategies, or more specifically MPC.

While traditional microcontrollers and processors are restricted to sequential computing, FPGAs are able to perform calculations in parallel. Can this be exploited in some way to increase the speed of MPCs?

To answer this, an MPC was implemented and tested on a simple system. By analyzing this implementation it was possible to say something about the time and resource utilization for MPC on FPGA in general.

In the final chapter of the report, the gathered information and experience was used to make conclucions regarding parallelization to improve MPC speed on FPGA, and what challenges one might face in such an implementation.

Also, the first chapter contains some basic information on PID-controllers in FPGA. This is of course not very groundbreaking stuff, but it allows for comparison of PID vs MPC in terms of lines of computer code needed, resources needed on the device, and last but not least; the end result.

2 PID-controller

2.1 Introduction

The PID-controller is one of most used controllers in the industry today. This has to do with the fact that such controllers are not very application specific, and are relatively easy to tune for acceptable performance. However, the PID-controllers are not capable of handling constraints, setpoint-handling is far from optimal, and if tuned poorly, may go into an oscillating and unstable state.

As a part of this thesis, a PID-controller was implemented on the FPGA. Of course, due to the simplicity of the PID-controller, there are no major advantages to using an FPGA instead of a microcontroller for this task as long as only one controller is needed. However, the true power of the FPGA lies in parallellism, and if several controllers are needed, this may easily be implemented without any loss of performance.

The PID-controller that was implemented required less than 100 lines of VHDL code and less than 1% of the capacity on the FPGA. A great thing about the PID-controller is that it doesn't need to deal with float integers. The floating point libraries take a lot of space on the FPGA, and they also require some clock cycles to generate results.

As a result, the PID-regulator is able to calculate a new output within less than 10 clock cycles.

2.2 Implementation

The implementation was done using integer variables. The only difficult operation that needs to be done in a PID regulator is division by sample time (because of the integrator and derivator functions). A neat trick to make this a simple task is to keep the number of samples/second to some power of two, making the division a simple bit shift operation.

As the development board that was used had a serial port, a driver for this was created. This allowed the user to enter controller values (Kp, Ti & Td), and also to set the reference from a terminal window on a computer.

The PID-controller was tested using reference jumps between 100, 200 and 250 rad/s in varying sequence. Due to the reference not being availiable in MATLAB, it is not displayed in the plots.

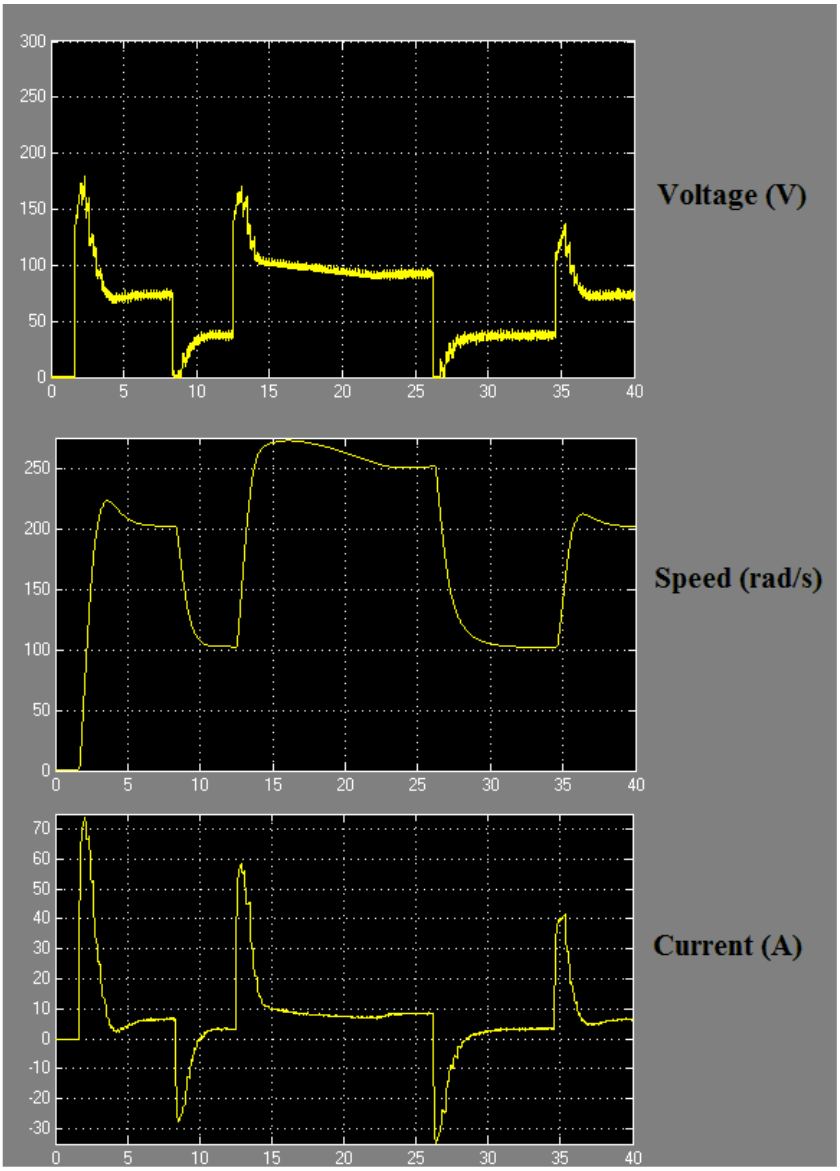

Figure 1: Plots for implemented PID

3 Model Predictive Controller (MPC)

3.1 Basic idea behind MPC

Model Predictive Control – as the name says – uses a model of the system to predict the systems future states, given some future calculated inputs. A QP problem is generated, that when solved will find the optimal control inputs that will minimize the difference between setpoints and system outputs.

The MPC horizon is typically somewhere between 5 to 50 samples. Even though the optimal control input is found for all these samples, it's common to use only the input calculated for the current sample, and then solving the QP-problem once again for the next sample. This allows us take into account differences between the model and the real process.

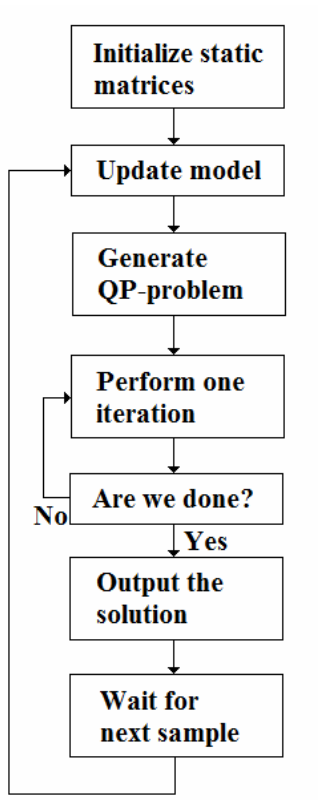

Figure 2: MPC state diagram

3.2 MPC problem formulation

The MPC can be formulated either as an LP (linear problem) or as a QP (quadratic problem). With both formulations, the objective is to minimize a cost function.

Example cost functions for MPC as LP and QP:

$$\text{LP: } f = \sum_{j=1}^{Np} \left\| x(k+j) - x_{REF}(k+j) \right\| + \sum_{j=0}^{Nu-1} \left\| \Delta u(k+j) \right\|$$

$$\text{QP: } f = \sum_{j=1}^{Np} \left\| x(k+j) - x_{REF}(k+j) \right\|^2 + \sum_{j=0}^{Nu-1} \left\| \Delta u(k+j) \right\|^2$$

Here Np and Nu are prediction and control horizons respectively. Clearly, the first part of the cost function aims to minimize the deviation between a state and its reference, while the second part minimizes changes in the control variable. There are different ways to set up the cost function, and one could for instance also minimize the use of the control variable, not just the derivative of it.

A comparison of MPC as LP vs MPC as QP:

Table 1: MPC defined as LP vs QP

Linear Programming	Quadratic Programming
Easy to solve	Harder to solve.
Possible non-unique solutions	Unique solution.
Difficult to tune	
Slow action when far away from desired state.	Much action when far away from desired state.
Much action and jumping when close to desired state.	Smooth action when close to desired state.

Due to these differences, Quadratic Programming is used almost exclusively for solving MPC problems.

3.3 Formulating MPC as a standardized QP-problem

In a time-discreet system, a model of the system can be described as

$$x(k+1) = A*x(k) + B*u(k)$$
$$x(k+2) = A*x(k+1) + B*u(k+1) = A^2*x(k) + A*B*u(k) + B*u(k+1)$$
$$x(k+3) = A*x(k+2) + B*u(k+2) = A^3*x(k) + A^2*B*u(k) + A*B*u(k+1) + B*u(k+2)$$

If this is continued, we end up with the following on matrix form:

$$\begin{bmatrix} x_{k+1} \\ x_{k+2} \\ \vdots \\ x_{k+n} \end{bmatrix} = \begin{bmatrix} A \\ A^2 \\ \vdots \\ A^n \end{bmatrix} * x(k) + \begin{bmatrix} B & 0 & \cdots & 0 \\ AB & B & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ A^{n-1}B & A^{n-2}B & \cdots & B \end{bmatrix} * \begin{bmatrix} u_k \\ u_{k+1} \\ \vdots \\ u_{k+n-1} \end{bmatrix}$$

Or on short form:

$$\vec{x} = \hat{A}*x(k) + \hat{B}*\vec{u}$$

We may now generate the standardized QP-problem, which is on this form:

$$\min_z 0.5*z^T*Q*z + c^T*z$$
$$Jz \le g$$

Where:

$$Q = \hat{B}^T*Q_x*\hat{B} + Q_u$$
$$c = \left(\hat{A}^T*x(k)^T - X_{REF}\right)*Q_x*\hat{B}$$

Qx and Qu are the weight matrices for the states and control inputs.

J is a matrix that when multiplied with z will give future control inputs, and changes to states due to control input. This is usually a negative and a positive identity matrix stacked over negative and positive B_hat.

g is the boundary matrix, containing minimum and maximum control inputs, and minimum and maximum allowable change to the states, due to the future control inputs. This is usually some fixed level for the control inputs, while allowable change to states due to control inputs are calculated with a combination of x(k), A_hat and the chosen boundaries.

z is the 1-D matrix containing the calculated (and optimal) control inputs.

3.4 Solving the QP-problem

There exist many algorithms that solve QP-problems. Some of these are:

- Gradient (Steepest descent).
- Interior Point.
- Active-set.

These algorithms are described in detail with pseudo code in [1]. Originally none of these are very efficient, but throughout time there has been developed several subversions of them which has greatly improved their performance.

When measuring the performance of such an algorithm, there are usually two criterias. These are the number of iterations needed for convergence and the computational cost for each iteration.

When looking at convergence speed vs computational cost it is generally the two last-mentioned algorithms, interior point and active set, that are mentioned most in the litterature, and accepted as “better” than steepest descent. This is most likely due to the fact that steepest descent often requires very many iterations.

In this implemention a version of the interior-point method [2,3] was chosen both due to its reputation as a fast algorithm and the availiabilty of code.

In [4], it is proposed a modified version of the active-set method, which is supposedly much faster than earlier versions, and this should be considered a possible candidate for later implementations.

3.5 Algorithm for “infeasible interior point method”:

The code for the infeasible interior point method can roughly be written as follows:

$$Z = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \lambda = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, e = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

$\mu = 1$

$while(\mu > \mu_{\max})$

{

$\Gamma = -\lambda^{-1} T$

$R_1 = -QZ - J^T \lambda - c$

$R_2 = -JZ + g - \sigma\mu\lambda^{-1} e$

$\Delta Z = (Q - J^T \Gamma^{-1} J)^{-1} * (R_1 - J^T \Gamma^{-1} R_2)$

$\Delta\lambda = \Gamma^{-1} R_2 - \Gamma^{-1} J \Delta Z$

$\Delta T = -T + g - J(Z + \Delta Z)$

$(Z, \lambda, T) = (Z, \lambda, T) + \alpha * (\Delta Z, \Delta\lambda, \Delta T)$

$$\mu = \frac{(Te)^T * \lambda e}{Nc}$$

}

Figure 3: Detailed code for "infeasible interior point method"

Notice that this algorithm is computationally quite expensive. However, it doesn’t require very many iterations, and this makes it a good candidate for parallelism as described in chapter 5.

4 Implementation of MPC on FPGA

4.1 Chosen approach

First, the controller was set up in MATLAB, using standard MATLAB notation and matrix operations. The “Quadprog” routine was used to solve the optimization problem.

Because there are no libraries for matrix operations on FPGA’s, the creation of such a library was considered. However, this turned out to be rather inefficient and complex due to the nature of matrix operations. One would have to account for different matrix sizes, their location in BRAM, all the different operations (addition, multiplication etc), and the fact that sometimes the transpose of the matrix is used. The greatest drawback though, would be the lack of control, resulting in wasted time cycles. This could be for instance if performing operations with diagonal or triangular matrices.

Because of this, it was decided to simply implement the matrix operations as series of (nested) for-loops, combined with some whiles and ifs, making the FPGA-code rather dirty and large (the finished code was a slightly more than 1000 lines). The MATLAB code was converted into this simplified form, which actually also increased the performance in MATLAB. The Quadprog routine was replaced with the interior point method.

This was then implemented on the FPGA, using VHDL. Note that this gives great possibilities for debugging on the different levels of abstraction. Comparisons to the Quadprog routine shows if the chosen QP-solver is working, and comparisons to the simplified MATLAB code allows for “finer” debugging.

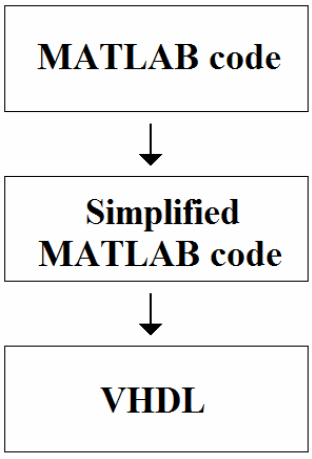

Figure 4: Different layers of abstraction for the MPC code

Because MPCs perform a lot of calculations requiring good accuracy, and the the fact that the controller is based on a model of a real system, it was chosen to use floating point integers. The IEEE standard 32-bit floating point integers with 8 bit exponent and 23 bit mantissa were chosen.

4.2 Storing the data

Due to the amount of data involved in an MPC-controller (or any huge problem for that matter) it is not possible to store the data in the FPGAs registers. For this given implementation, the Block RAM within the FPGA was used.

To make accessing of the variables as easy as possible, each matrix used in the MPC-routine was given its own slice of the BRAM. Single precision floats are 32 bit, so each BRAM slice would need its own 32-bit data bus. The size of the address-bus naturally depends on the size of the matrix. For a matrix of size 30*6, the address bus would be 8 bit. The first 5 bits would represent the row in the matrix, and the last 3 bits would be the column. This would allow for a matrix of size 32*8.

An example of addressing one of the matrices, taken from the implementation:

```
Q_addr <= conv_std_logic_vector(i, Q_r) & conv_std_logic_vector(j, Q_c);
```

Here, Q_r is some constant integer saying how many rows the matrix has, and Q_c is the number of columns. “i” and “j” are iteration variables, iterating over the rows and columns.

This way of storing matrices in the BRAM is very practical, but not very efficient. If the number of columns and rows is barely more than some power of two, one would in the worst case use almost 4 times more BRAM than necessary. On average, one would use almost twice the needed amount. To avoid this, one could create a block working as a memory allocator. However, this would probably mean spending more clock cycles for each memory access, which is much worse than spending extra memory.

Giving each matrix its own slice of the BRAM is also crucial when considering the time needed to access the data during run-time. Separate slices allow us to fetch data from each slice during the same clock cycle.

4.3 Testing the MPC implementation

The MPC-algorithm was implemented on the Spartan-3E development board. This board has a DAC, which in turn was connected to a NI-DAQ board and fed to the computer through USB. A simple UART protocoll was used for debugging, and for entering set-points. This used the RS-232 port on the development board, and was fed to the computer using a RS232-USB adaper.

To test the MPC, a model of a simple DC motor was used. This model was running in simulink, and the FPGA should control it. A DC motor only has two states (current and rotiational speed), and only one input (voltage), but should clearly be able to demonstrate the functionality of the controller. The MPC-code was written on such dynamic form that expanding the system to contain more states/control outputs should be just to modify some global parameters.

The implemented MPC had a prediction horizon of 10 samples, and a sample time of 50 ms.

The following constraints were put on the system:

Table 2: Constraints on the implemented MPC

	Minimum	**Maximum**
Current (A)	-10	40
Rotational speed (rad/s)	0	1000
Control input (V)	0	200

The system was tested with a sequence of reference jumps, spinning the motor up and down, varying the set-point between 100, 200 and 250 rad/s. The actual reference is not included in the plots though, as this was not a part of the data in Simulink.This led to the following resulting plots:

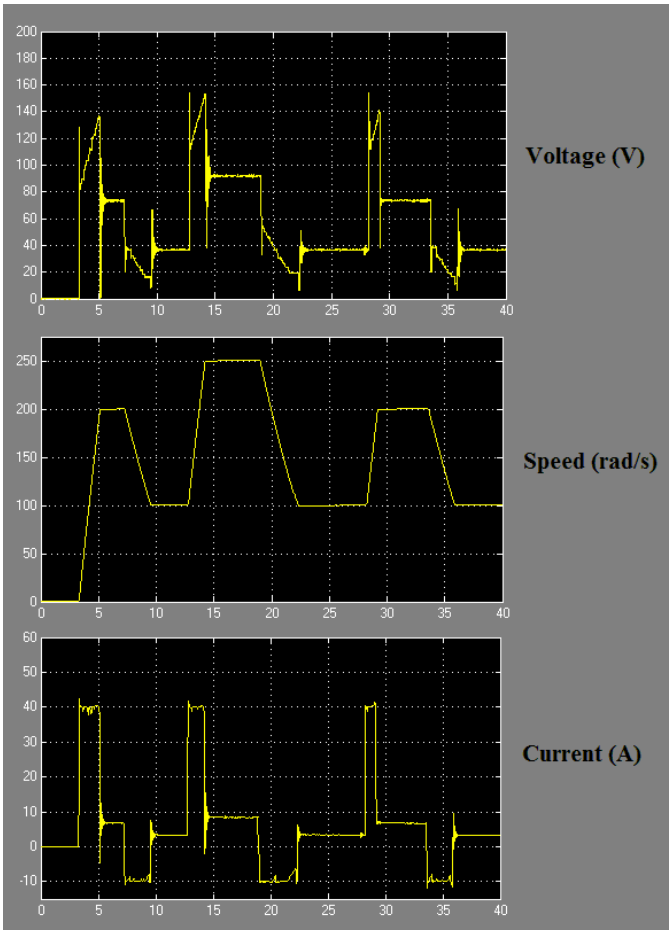

Figure 5: Plots for implemented MPC

As we can see, the controller is fast and efficent, keeping the system within the specified boundaries and following the set-point very good. In this test all the minimum constraints were at some point active, and also the maximum constraint for the current.

Zooming in on the voltage, we can clearly see that the MPC is working with a 50 ms sampling time:

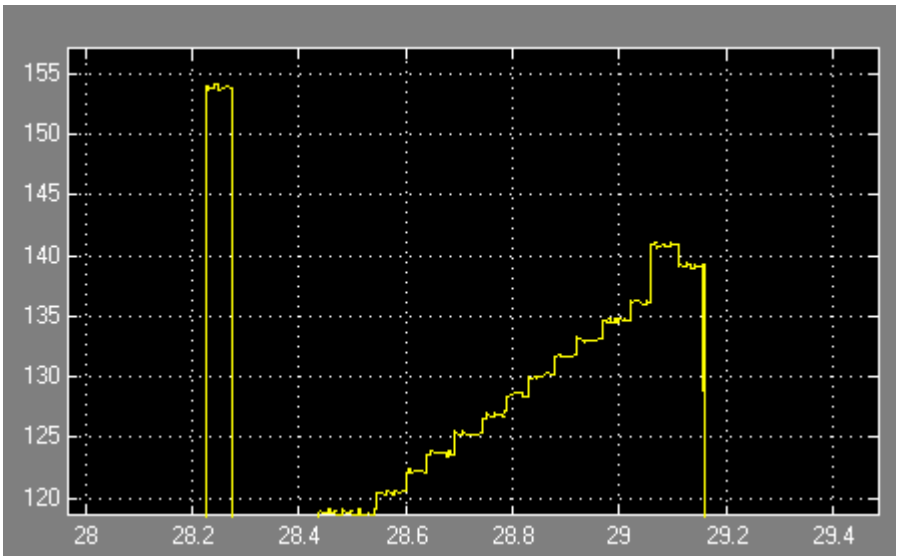

Figure 6: Zoomed view showing 50ms sampling time

When comparing the result from the MPC versus the PID, it is quite obvious why MPC should become a more common controller strategy. Setpoint-handling is very much better, power dissipation in the system is reduced and the system is generally more stable.

On the downside is the added complexity, meaning increased costs both to develop and maintain. The largest downside though, is probably the reduced sampling rates when compared to for instance the PID-controller.

4.4 Clock cycle usage

To identify the time needed to solve the QP-problem, simple counters were inserted in the MPC code.

Table 3: Approximate number of clock cycles needed for MPC in FPGA

Number of iterations needed:	6-15
Number of clock cycles/iteration	100 000
Number of clock cycles needed pr sample (includes updating model etc)	620 000 - 2 000 000

This shows that 50 ms sampling time, as used in this case on a 50 MHz FPGA, is about as fast as it can go. Note that this implementation was not very optimized in any way. There were many delays that could have been removed, and there has not been put any particular effort into finding optimal settings for the QP-solver. This means that there is probably a lot to gain in both number of iterations needed and time pr iteration. Reduction of more than 20% in total clock cycles should not be impossible with some work, but because of extremely long time to synthesize the code (> 20 min) this was ignored for now.

As table 3 shows, the number of iterations varies a lot. When the controller is stable and settled it is much faster than when it needs to do a lot of controlling. This is a little bit unfortunate though, because it is clearly when the process is far from it's setpoint that we need to be able to control it quicky. There are several actions to choose from if the controller should happen to not finish in time:

1. Abort the algorithm, output whatever has been calculated so far
2. Abort the algorithm, output the same as last sample and try again
3. Continue the algorithm, leaving the current output as is. Try to "catch up".

In this implementation the last point was chosen as it was simply the easiest to implement.

Another important issue is how the time usage scales with an increased problem. This can easily be seen from the loops in the code.

An estimate shows that the algorithm implemented has a running time of $O(c_1 m^3 + c_2 n^2)$. The m^3 part comes from the fact that a matrix of size (Nu*Np,Nu*Np) has to be inverted, and there is one calculation where two 2-d matrices are multiplied with each other. The n^2 part basically comes from each time a 2-d matrix is multiplied with a 1-d matrix. There are a lot of these multiplications in the used algorithm, thus c_2 is much larger than c_1.

4.5 Resource usage in the FPGA

The FPGA on the Spartan 3E development board is of the type “xc3s500e”. This is definitely an FPGA on the smaller end of the scale, and as such is probably not the best choice for an MPC implementation.

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	1,887	9,312	20%
Number of 4 input LUTs	8,487	9,312	91%
Logic Distribution			
Number of occupied Slices	4,594	4,656	98%
Number of Slices containing only related logic	4,594	4,594	100%
Number of Slices containing unrelated logic	0	4,594	0%
Total Number of 4 input LUTs	8,876	9,312	95%
Number used as logic	8,483		
Number used as a route-thru	389		
Number used as Shift registers	4		
Number of bonded IOBs			
Number of bonded	39	232	16%
Number of RAMB16s	20	20	100%
Number of BUFGMUXs	3	24	12%

Figure 7: Device Utilization Summary

As figure 6 shows, 98% of the FPGA was used, as well as all the 20 Block RAMs (one block for each matrix). However, the space needed for an MPC implementation does not increase much with increased problem size (more states, constraints etc). This means that with a larger FPGA, even though one also could have a larger problem, one could use the extra space to add parallellism.

This summary doesn’t specify how many bytes of the BRAM are being used, but counting the size of the matrices gives us 4336 elements (address locations).
$4336 * 32 \frac{bit}{element} = 138752 bits$. The chip contains 360 kbit BRAM, so there seems to be some space left over. Of course, most of the matrices will grow quadratically with respect to the problem size. This is one of the big reasons why this specific FPGA is not recommended for MPC. There are FPGA’s out there with a lot more BRAM than this one, often several megabytes.

One trick however, if one has to little BRAM, or too few multipliers (restricting the number of BRAM partitions) would be to reuse the same BRAM several parts in the program. It seems that performing arithmetics with simple loops leads to a lot of temporary matrices being needed. Once their data has been used, they can simply be overwritten by some new data and used later in the program.

5 Improving performance on the FPGA

5.1 Parallelism

The major advantage of FPGAs vs regular microcontrollers is the ability to perform several tasks in parallel. There are several ways this can be exploited to achieve better performance/speed in an MPC implementation.

Using the BRAM on the FPGA gives us some degree of “free” parallelism, when compared to typical usage of one external RAM unit. This is because the BRAM can be divided into separate RAM parts, with their own address- and databuses. This makes it possible to retrieve/write data to/from several matrices simultanously. This can easily reduce the number of clock cycles needed with 20-25%.

The speed of one algorithmic operation (mainly multiplication) is of course important. In the world of microcontrollers, specifically the AVR series, this is usually done in 2 clock cycles as long as we are using the specified amount of bits, meaning 8-bit multiplication for 8-bit AVRs, 16-bit multiplication for 16-bit AVR etc. With the floating point package in Xilinx, the user can choose the speed vs area used on chip. According to the documentation on the package, doubling the speed almost quadruples the amount of resources used, meaning that one could possibly achieve higher throughput by using slower arithmetics as this would allow for many more units, giving us the possibility to perform a lot of operations in parallel.

For the FPGA used in this project, the xc3s500e, a multiplication device using 2 clock cycles uses almost 7% of the total number of slices, while an adder uses roughly 2%.

One of the main issues for parallellism is “What can be done in parallel?” All QP-solvers are iterative. These iterations are of course dependant on each other, and as such will have to be done sequentially. Therefore the objective should be to make each iteration as efficient as possible.

5.2 Breaking down matrix operations into parallel actions

Consider the matrix multiplication $A = B * C$ where B and C are of sizes NxM and M respectively. This operation can be written as:

$$A_i = \sum_{j=1}^{j=M} B_{i,j} * C_j$$

Figure 8: Matrix multiplication

This operation would require N*M multiplications and give us a running time of $O(n * m)$ when done sequentially.

Now what if we use M multiplicators and adders to calculate each element of A in parallell? This could clearly scale down the problem with one degree, giving us a running time of $O(n)$. The same could of course also be done when multiplying two 2-d matrices, reducing running time from $O(n^3)$ to $O(n^2)$

Using this method for the QP-solving algorithm would have tremendous effects, reducing the running time of the algorithm from $O(c_1 m^3 + c_2 n^2)$ to $O(c_1 m^2 + c_2 n)$

Most of the matrices in the implementation are of some size related to Np*Nu, the number of prediction samples and the number of outputs. A prediction horizon greater than 10 is rarely needed, and more than 3 outputs would definitely be called a large problem. This would translate to 30 multiplicators and adders. Using arithmetic units at maximum speed, this would roughly use 3 times the space availiable on the FPGA used in this implementation. However, using a larger FPGA, and maybe reducing the speed of the arithmetic units some, this should be possible to implement.

There is a different problem that arises though. Note that all these arithmetic units would work with the same matrices. This means that they would all try to access the same memory at the same time, which would not work. A far-fetched solution could be to have the same amount of copies of all the the matrices as the number of arithmetic units. This would however require an enormous amount of memory, and this basically leaves this method of parallelism unusable.

5.3 Several matrix operations in parallell

Another, more realistic approach to the parallellism issue, could be to calculate several parts of the QP-algorithm in parallell as depicted in figure 7:

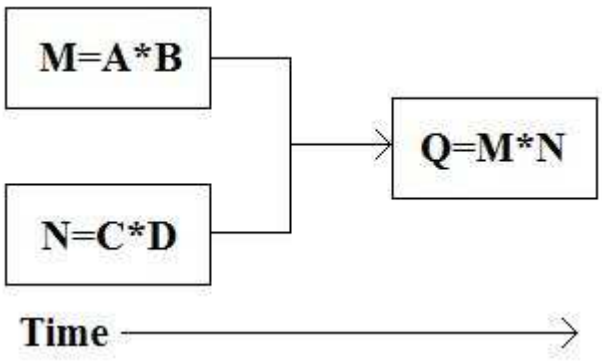

Figure 9: Parallell multiplication

If we look at the running time of the implemented algorithm, $O(c_1 m^3 + c_2 n^2)$, the main goal of this approach would be to reduce c_2. Considering that c_1 is already quite small, this should give some very noticable improvements to the running time.

In this approach, the same problem could be encountered as in (5.2) if the same matrix is used is several operations. However in this case one would not need nearly as many copies of the matrices to get around the issue. Also, some performance gain could definitely be achieved without addressing this issue at all, leaving these as sequential operations.

By looking at the code for the implemented algorithm, one can easily see many matrix operations that would be suitable for performing in parellel, and reductions of running time could easily be as much as 50-60%. Even more if one is able to cut down the time needed for the matrix inversion.

5.4 Finding an efficient QP-solver

Usually when looking for an efficient QP-solver, the only criterion is how fast it is. When trying to utilize the parallelism aspect of the FPGAs another criterion arises: how parallelizable it is.

As iterations in the QP-algorithm have to be done sequentially, the number of these should clearly be kept to a minimum.

Most QP-solvers needs to invert some matrix, and in general, inverting an M*M matrix requires O(M^3) calculations [6]. In this implementation, the matrix inversion was done using Gaussian Elimination. Another much used way to invert matrices is by LU-decomposition, but this should result in roughly the same amount of calculations needed as the Gaussian Elimination method.

Note also that O(M^3) is the same number of calculations as is needed when multiplying a M*M matrix with another M*M matrix.

In [5] it is also conluded that matrix-inversion by Gaussian Elimination can be done faster by using parallel processing. However, the paper also states that for an NxN matrix, “we would like to have parallel transfer capability for n numbers”. This leads us back to the same problem as in (4.2).

This means that for a QP-solver to be efficient, it should avoid inverting matrices, and avoid multiplying two 2-D matrices with each other.

6 Conclusions

6.1 Results

It has been shown that implementing MPC on FPGA is feasible and that there are many things that influence the achieved speed:

- MPC size, mainly the number of outputs and prediction samples
- Choice of QP-solver
- FPGA size
- Choice of speed vs size for arithmetic units
- Choice of integer size

Clearly it's hard to give a presise answer to such questions as "how big FPGA is needed for this or that MPC problem". However this report should give the reader some ideas on both how to implement MPC on FPGA, and how this can be made as efficient as possible.

Specifically the report shows how parallelism can be implemented, and that this can reduce the running time by a significant factor. It's shown that even though one can use several arithmetic units, there is a possible bottle neck in the way one accesses data from BRAM which may have to be addressed in some way.

It seems that with todays FPGAs we are quite close to the "barely possible" level when it comes to MPC and QP-solvers. However, as time goes by, both the size and speed of FPGAs are likely to increase a lot, and I'm certain that MPC on FPGA will become quite common in many industrial applications.

One of the objectives of this thesis was to create function blocks for PID and MPC controllers. Both of these have been created with great success, which can be seen by the plots from MATLAB where the controllers were tested on a simulated DC motor. These plots clearly show the difference in result from the two types of controllers.

6.2 Further studies

As this report has shown, the main problem when trying to utilize the parallelism aspect of the FPGA lies in the nature of the QP-solver. Thus it should be looked into finding a QP-solver that matches the criterias listed in (5.4).

It would be interresting to see an actual implementation of MPC using parallelisation, and see numbers on the actual performance gains compared to a sequential implementation.

7 References

[1] Nocedal, J., Wright, S.: *Numerical Optimization*. Springer Series in Operations Research. Springer, 2006.

[2] Wright, S.: *Applying New Optimization Algorithms to Model Predictive Control.*

[3] Ling, K.V., Yee, S.P., Maciejowski, J.M.: *A FPGA Implementation of Model Predictive Control.* 2006.

[4] Milman, R., Davison, E.J.: *A Fast MPC Algorithm Using Nonfeasible Active-Set Methods.* Springer, 2008.

[5] Pease, M.C.: *Matrix inversion using parallel processing.* Journal of the ACM, October 1967

8 Appenix

8.1 List of figures

8.2 List of tables

8.3 Content on CD

This report should be accompanied by a CD containing:

- This report in digital form
- VHDL files for PID regulator with VGA driver
- VHDL files for MPC regulator
- MATLAB code for MPC regulator
- Tutorial: Getting started with VHDL