Circuits and Systems
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ens.ewi.tudelft.nl/

CAS-MS-2012-01

M.Sc. Thesis

A FPGA implementation of a real-time inspection system for steel roll imperfections.

Martin Molenaar B.ICT

Abstract

Today's production processes are more and more optimized to be competitive. The production demands are increased for speed and quality. These increased demands do not pass the roll shops in the steel industry. In the roll shop periodically the rolls from the rolling mill are checked for imperfections. The imperfections are detected by special inspection systems. Improving the inspection systems can speed up the overall process significantly in the roll shop.

The request for an improved inspection system results in a new generation inspection system. This inspection system should measure more signals at the same time and process the signals faster. To achieve this result the measurements are digitalized and processed in parallel on a FPGA. Speed and quality demands are also asked from the engineers by designing and maintanance of the inspection system.

In this thesis a High-Level Synthesis tool is selected to implement the mathematical model of the inspection system. The tool selection is done based on a comparison between three HLS tools, namely: CatapultC, ROCCC and Compaan. For this implementation Compaan is the most promising one. Compaan is able to split the data streams processing in concurrent systems with distributed memories. With Compaan as development tool the main part of the mathematical model is implemented in four months. This is four times faster than the preceding implementation.

A FPGA implementation of a real-time inspection system for steel roll imperfections.

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Computer Engineering

by

Martin Molenaar B.ICT
born in Amsterdam, The Netherlands

This work was performed in:

Circuits and Systems Group
Department of Microelectronics & Computer Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

This work was sponsored by:

LISMAR

Lismar Engineering B.V.
Ambachtenstraat 55
1191 JM OUDERKERK AAN DE AMSTEL
The Netherlands

Delft University of Technology

Delft University of Technology
Department of
Microelectronics & Computer Engineering

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled **"A FPGA implementation of a real-time inspection system for steel roll imperfections."** by **Martin Molenaar B.ICT** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: January 26, 2012

Chairman:

prof.dr.ir. A.J. van der Veen, Circuits and Systems, TU Delft

Advisors:

dr.ir. T.G.R.M. van Leuken, Circuits and Systems, TU Delft

ir. C.M.J. van den Elzen, NDT Specialist, Lismar Engineering BV

Committee Members:

dr.ir. A.J. van Genderen, Computer Engineering, TU Delft

dr.ir. A.C.J. Kienhuis, CEO, Compaan Design BV

Abstract

Today's production processes are more and more optimized to be competitive. The production demands are increased for speed and quality. These increased demands do not pass the roll shops in the steel industry. In the roll shop periodically the rolls from the rolling mill are checked for imperfections. The imperfections are detected by special inspection systems. Improving the inspection systems can speed up the overall process significantly in the roll shop.

The request for an improved inspection system results in a new generation inspection system. This inspection system should measure more signals at the same time and process the signals faster. To achieve this result the measurements are digitalized and processed in parallel on a FPGA. Speed and quality demands are also asked from the engineers by designing and maintanance of the inspection system.

In this thesis a High-Level Synthesis tool is selected to implement the mathematical model of the inspection system. The tool selection is done based on a comparison between three HLS tools, namely: CatapultC, ROCCC and Compaan. For this implementation Compaan is the most promising one. Compaan is able to split the data streams processing in concurrent systems with distributed memories. With Compaan as development tool the main part of the mathematical model is implemented in four months. This is four times faster than the preceding implementation.

Acknowledgments

I would like to thank some people. They helped me on my way in writing this thesis:

At first I want to thank my advisor: dr.ir. T.G.R.M. van Leuken, Delft University of Technology. Rene spent much time in giving me feedback. I was able to improve my thesis thanks to his coaching.

Secondly I want to thank Lismar Engineering BV, particularly:

- ir. C.M.J. van den Elzen (NDT Specialist). Elmar shared to me a lot of his knowledge about the backgrounds of Eddy Current and the mathematical model.
- ing. D.C. ter Haar(Hardware Engineer). Daan and I discussed the hardware related topics many times.

Thirdly my thanks are for Compaan Design, especially: dr.ir. A.C.J. Kienhuis (CEO). Bart gave me much technical support.

Finally I want to thank my family and friends for their mental assistance.

Martin Molenaar B.ICT
Delft, The Netherlands
January 26, 2012

Contents

List of Figures

List of Tables

1 Introduction

1.1 Context

Lismar Engineering B.V. (Lismar) is world market leader of roll inspection systems. Lismar is developing a new generation Eddy Current (EC)[1] inspection systems.

With the new generation EC inspection systems Lismar will improve six important aspects of the inspection, namely: speed, sensitivity, repeatability, quality, classification and flexibility. The *speed* will be improved to finish the inspection in shorter time. The *sensitivity* will be improved to find smaller cracks and better separate cracks from noise. The *repeatability* will be improved to get fewer differences between the results if the same roll is scanned multiple times. The *quality* will be improved to return the same value independent to the length and rotation of the crack (only the depth is important). The *classification* will be improved to separate cracks, bruises and magnetic fields from each other in a better way. The *flexibility* will give the opportunity to tune the inspection system for every roll type and grinding program.

Figure 1.1: inspection system

To achieve the improvements there are two main changes between the current EC inspection systems and the new generation EC inspection systems. The current EC inspection systems use one channel and process this channel with analog electronics. The new generation EC inspection systems will use twenty-four channels and processes the data digitally. From now the term "inspection system" will reference to "the Lismar new generation EC inspection system".

1.2 Inspection system

In the block diagram of Figure 1.2 the inspection system (light grey block) is drawn and the terminal connected to the system to display the results. Inside the system there are three main parts: *input* (purple block), *mathematical model* (dark grey block) and *network* (green block). The *input* of the system is the twenty-four coils array. From each coil there are samples coming with a speed of 10MHz. The *mathematical model* contains three sub-blocks to process the input, namely: *demodulation*, *filters* and *finalization*. The *demodulation* extracts measurement results from the alternating

[1] More about Eddy Current method in Section 2.3

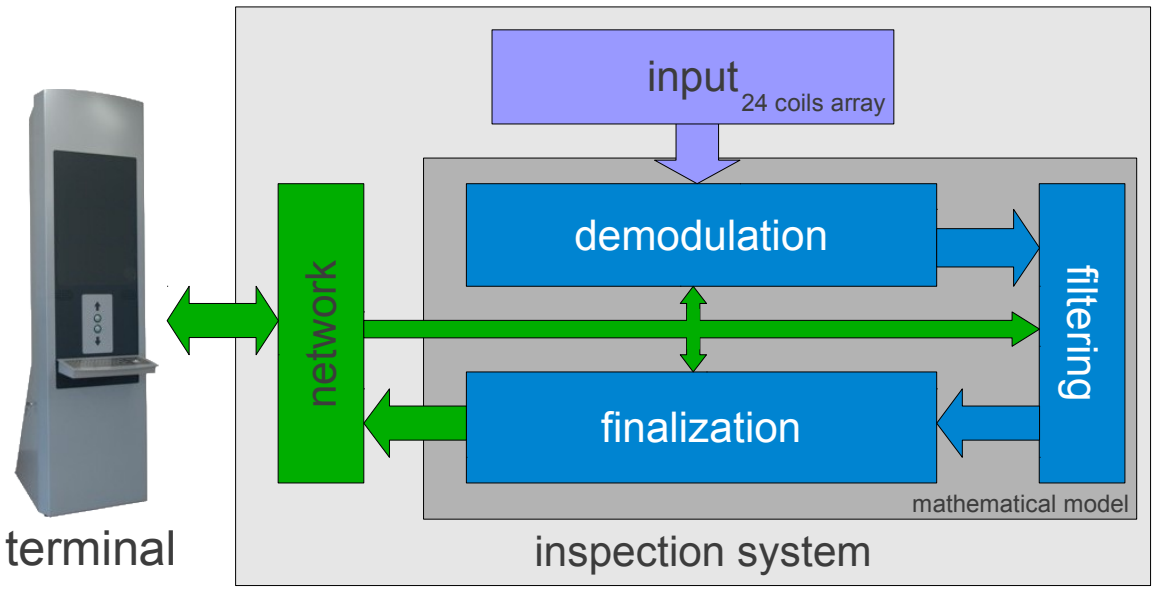


Figure 1.2: Block diagram inspection system.

current carrier wave of the coils. The *filter* removes from the results measurement noise. The *finalization* post-processes the filter results. A part of the finalization is rectification. The *network* transports system settings and (intermediate) results between the inspection system and the terminal by Ethernet.

The inspection system will process the data real-time. Real-time processing is required because of the high bit-rate. This high bit-rate can not be handled in a general purpose micro controller nor a Digital Signal Processor, therefore the inspection system is implemented in a Field-Programmable Gate Arrays (FPGA)[20]. Implementing the design in a FPGA sounds nice in terms of speed and performance. But the complexity is much higher than a micro controller implementation, because of the fact that designing, testing and debugging are more complex.

Lismar started designing the systems two years ago. At this moment a working prototype of the hardware is already finished. The communication between the terminal and the FPGA has been implemented and tested too. The mathematical model is already designed and simulated in Matlab[2] by the NDT specialist of Lismar . The implementing of the mathematical model on the FPGA is still in progress. This implementation is referenced in this thesis as " Lismar implementation".

1.3 Problem definition

The current implementation method of the Lismar implementation is done on register level in a Graphical User Interface. Specific knowledge about the implementation is

[2]MATLAB® is a high-level language and interactive environment that enables you to perform computationally intensive tasks[11].

required to build, maintain and extend the system. To be more competitive Lismar wants a more efficient way to map the mathematical model of inspection system to hardware to decrease the time to market. Lismar asked Delft University of Technology to research for a more effective and accessible way to implement the model. The implementation and maintenance of the model has to be efficient and cost-effective (Lismar is a small company with 30-40 employees).

Lismar already decided to use a FGPA, designed the hardware and established the communication between the terminal and FGPA. Therefore this is not part of this research. Also the improvement of the mathematical model is out of the scope.

1.4 Solution and contribution

This thesis describes the research for a more effective implementation approach of the mathematical model. To find this approach a number of available tools are selected. From this tools the three most promised ones are selected for benchmarking, namely CatapultC, Compaan and ROCCC. The benchmarking is done with a time critical part of the mathematical model. After benchmarking, one tool is selected to implement the whole mathematical model. This implementation of the mathematical model is criticized and compared with the results of the Lismar implementation.

The best results are achieved with a hybrid solution of Compaan and a second tool. Compaan is a commercial development tool which analyzes the data stream of a system and splits the data stream in concurrent systems connect with distributed memories. Almost all concurrent systems are generated by Compaan, but some computational concurrent systems have to be created outside Compaan. These computational systems can be kept as simple as possible by using Compaan in the right way.

By using Compaan the main part of the mathematical model is implemented in four months. The final implementation contains the demodulation and eight filters. The implementation is compared and validated with the mathematical model created in Matlab. Several helper functions are written to connect the network part with the Compaan system.

The contributions achieved with this thesis:

- Critical analyze of the Lismar implementation.
- Benchmarking of three High Level Synthesis tools.
- Selection of an efficient implementation method.
- Implementation of the mathematical model with the selected tool.
- Memory optimisation.

1.5 Outline

The thesis contains the next outline. In Chapter 2 the basics are explained about roll grinding and Eddy Current Testing (ECT). These basics about roll grinding and

ECT are explained to understand overall working of the inspection system. Chapter 3 starts with a description about the current implementation method of Lismar . The current implementation method is not effective. Therefore high-level synthesis and development tools are discussed and three tools are selected. In Chapter 4 these three tools selected are benchmarked and scored. The tool with the highest score is selected for the final implementation described in Chapter 5. The final results of the implementation are reviewed in Chapter 6. Finally Chapter 7 gives a conclusion and some recommendations.

1.6 Confidential

The inspection system designed by Lismar is their intellectual property. Therefore the details are not discussed in the public part of this thesis. The overview is necessary to understand the basics of the thesis and to give a feeling about the complexity. Two parts are marked as confidential, namely Chapter 5 and Appendix C.

2 Steel roll inspection

Steel rolls are used for sheet rolling and are exposed at extremely high mechanical and thermal loads. Therefore small defects (like cracks or soft spots) can arise in the roll surface. If these defects are not found in time, they may grow faster and rolls can break or even explode (see Figure 2.1). Such an accident is potentially dangerous for the human beings in the neighborhood and often results in enormous economic costs (€100.000,- to €1.000.000,-). Therefore the roll is periodically removed from the rolling mill and checked in the roll shop for cracks. Depending on the roll type, rolls are removed after 15 minutes up to 6 weeks of continuous working.

Figure 2.1: A damaged roll in front.

2.1 Grinding process

If the used roll is in the roll shop the next three basic steps are done in generally. First the roll is cleaned by grinding a few tenths of millimeters from the top layer several times. This top layer is always damaged because of the intensive use. Secondly the system is scanned for cracks. If there are crack indications above a threshold the system will fall back on the first step otherwise the system will continue. If the crack is not removed after a few times of grinding the process manager can decide to send the roll to the lathe. At the lathe it is possible to remove the surface layer. Finally if all cracks are removed the required profile and roughness are ground in the roll.

2.2 Non-Destructive Testing methods

Scanning for cracks is done with Non-Destructive Testing (NDT) methods. Like the name suggests, NDT is a method of testing materials without causing damage to the material. The current inspection systems of Lismar contains Ultrasonic Testing (UT) and/or Eddy Current Testing (ECT). With UT it is possible to detect internal flaws in the roll, while ECT finds defects on the surface and just below, starting with a depth of 0.1mm.

The inspection system implemented in this thesis only uses the ECT method. The new generation inspection system will not replace the current inspection system. The current inspection system will still be produced and maintained.

2.3 Eddy Current Testing

By ECT an alternating current is applied to an inductor (such as a copper wired coil) which is positioned near the surface of the inductive material (in our case the mill roll). The alternating current generates a changing magnetic field below the inductor. Because the inductor is placed near the roll, the changing magnetic field introduces an alternating current in the roll, called Eddy Current. If there is a crack in the roll the Eddy Current is disturbed and the phase and intensity of the Eddy Current will change. The disturbed Eddy Current introduces a changed magnetic field which changes the alternating current in the driver inductor.

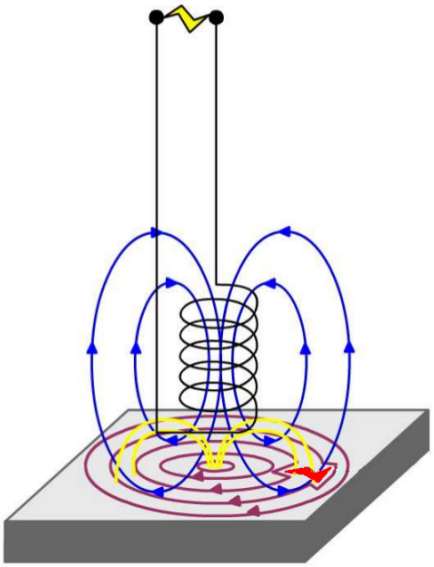
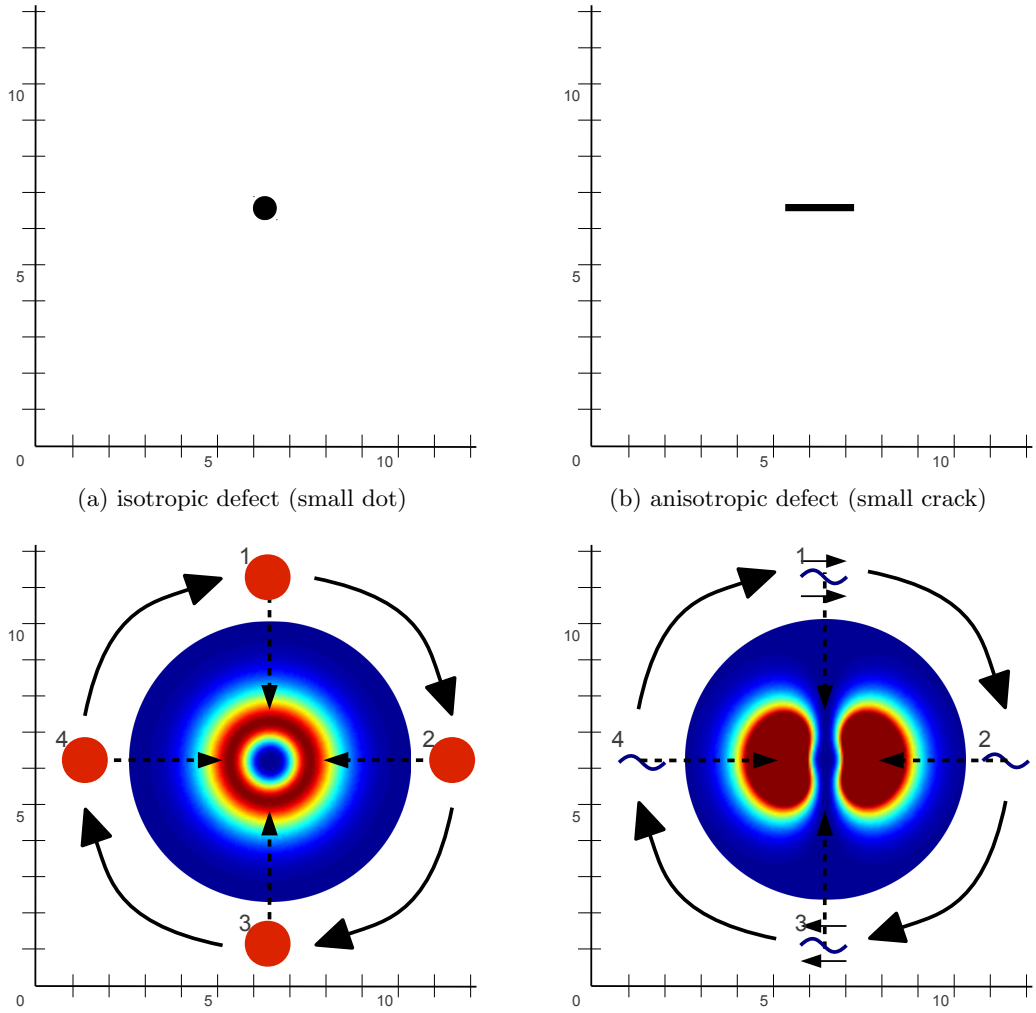


Figure 2.2: Coil used for Eddy Current Testing[15].

In Figure 2.2 there is a simplified model. The blue lines are the alternating magnetic fields from the coil. The magenta lines are the Eddy Currents. The yellow lines are the reversed magnetic fields generated by the Eddy Currents. At the bottom the inductive material, with a crack on the right side (red).

Figure 2.3 shows two Eddy Current response graphs. On the left a small dot (isotropic defect[1]) and on the right a small crack (anisotropic defect). The coloured circle in the middle is the 3D-graphs with rounded corners. The data to draw the 3D-graph is gathered by moving (scanning) the coil over the defect, from left to right, top to down. The colors represent the response values. A big color change means a big disturbance/response of the Eddy Current. Four positions are marked (numbered 1 up to 4). The dotted arrows point to the respective place in the graph. The solid arrows indicate the Eddy Current direction. Both defects are 'small'. This means that the diameter of the coil is a few times bigger than the defect.

[1] An isotropic defect responses in all directions on the same way.



(a) isotropic defect (small dot)

(b) anisotropic defect (small crack)

(c) response graph for isotropic defect

(d) response graph for anisotropic defect

Figure 2.3: Eddy Current response graphs

The response graph of the Eddy Current for the isotropic defect (see Figure 2.3c) is like a donut. When the coil moves over the defect, there is a high response if the defect crosses the Eddy Current below the coil. Marked position 1 up to 4 indicates high disturbance. If the defect is exactly under the middle of the coil there is no disturbance because the crack is to small and the Eddy Current is just around it.

The response graph of the Eddy Current for the anisotropic defect (see Figure 2.3d) is like two kidneys. At the marked positions 2 and 4 the EC disturbed much because the crack is perpendicular to the Eddy Current. At the marked positions 1 and 3 the crack is parallel to the EC and can pass like undamaged material. Like the isotropic defect there is hardly no response if the defect is exactly under the middle of the coil.

2.4 Conclusion

In this chapter three topics are described, namely *the grinding process*, *Non-Destructive Testing (NDT) methods* and *Eddy Current Testing (ECT)*. *The grinding process* is described to show the environment in which the inspection systems are used. The inspection system uses the *ECT* (a *NDT* method) to inspect the roll. Appendix C contains an additional description with more response graphs.

3 Background

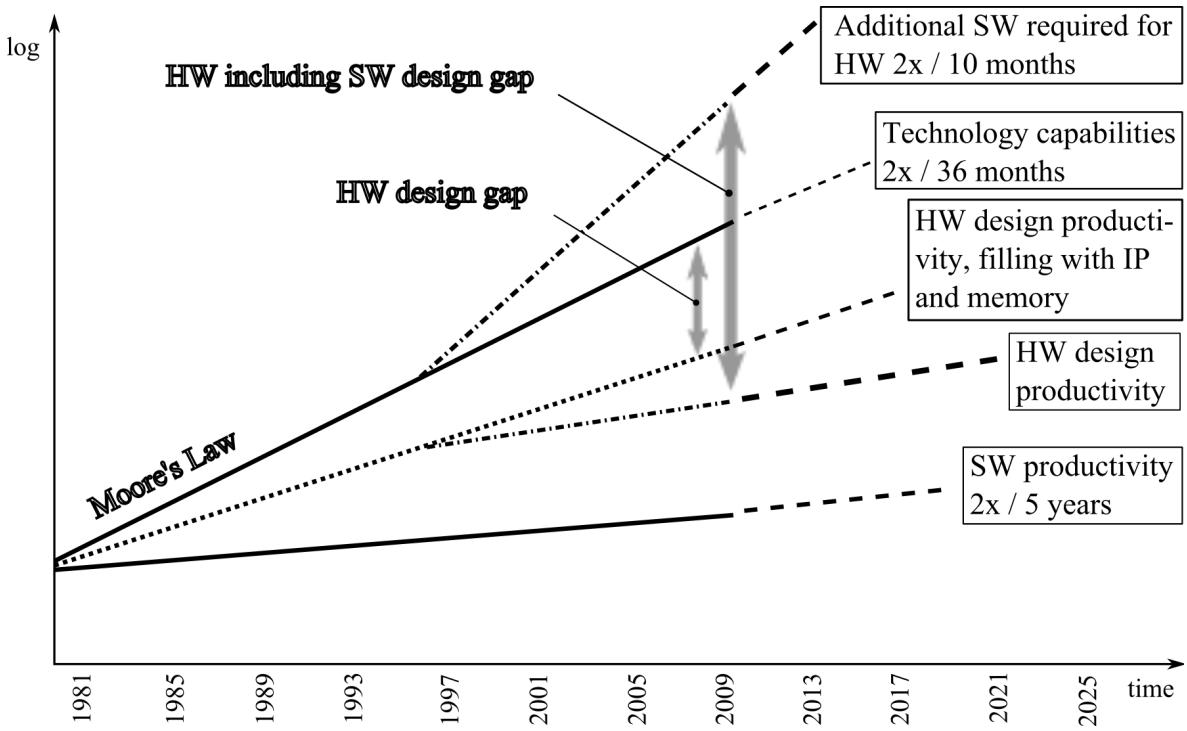


Figure 3.1: Hardware and Software Design Gaps versus Time[2].

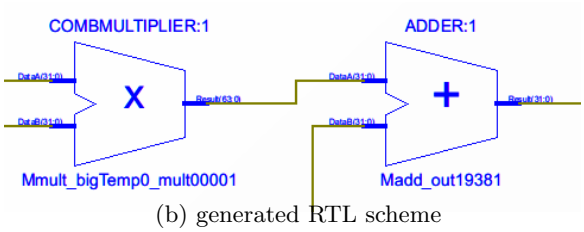
The Moore's Law is cited very often in researches. Gordon E. Moore the co-founder of Intel describes the "law" in his paper in 1965[14]. He describes the expected growing number of transistors (two times every 36 months) that can be placed on an integrated circuit for a reasonable price, later on this is called a "law" by Caltech professor Carver Mead. Four decades later we can conclude that his law is almost right until now. Beside challenge for the hardware engineers to design the new chips a new problem is coming up. Is it still possible to design an optimal program to use the hardware efficiently for engineers? In the International Technology Roadmap for Semiconductors editions 2009[2] the Moore's Law in combination with the hardware design productivity is placed (see Figure 3.1). The graph clearly shows the new problem of the 21^{th} century. The gap between the physical hardware and the hardware designs is growing.

3.1 High-Level Synthesis

To reduce the hardware design time, High-Level Synthesis tools are developed. This upcoming market[10] is growing and improving. High-Level Synthesis tools are tools

```
1 void MAC(int num1_in, int num2_in,
2          int num3_in, int &sum_out)
3 {
4   sum_out = (num1_in * num2_in) +
5             num3_in;
6 }
```

(a) C-code (high-level programming code)



(b) generated RTL scheme

Figure 3.2: High-Level Synthesis example

to synthesize High-Level programming languages into Register-Transfer Level (RTL). The advantage of a High-Level programming language is the strong abstraction. This means that the programmer only describes the functionality of the program and as less as possible the platform dependencies and the detailed implementation. The synthesizer should add details and translate it for the specified platform.

In Figure 3.2a there is an example C-code of the function MAC with three inputs and one output. Input *num1_in* and *num2_in* are multiplied together and *num3_in* is summed up with the multiplication result. A possible HDL implementation is showed in Figure 3.2b. This design is ready in one clock cycle. But it is also possible to create a (pipelined) design with two stages (one for the multiplication and one for the addition), because the synthesizer is not able to read that from the source code. This is one of the problems why High-Level Synthesis is very difficult. Note that High-Level Synthesis is not only done for C-code, but for more High-Level programming languages.

3.2 Related work

Lismar already starts to implement the mathematical model of the inspection system. For this implementation Simulink[1] is used, with a special plug-in. The High-Level synthesis plug-in for Simulink is designed to implement Matlab models on Xilinx FPGAs (the plug-in is part of the Xilinx®System Generator for DSP[13] software). Simulink is chosen because the computations of the mathematical model are already designed and tested in Matlab. With this plug-in it is possible to add and link hardware blocks in a graphical way together. There are several types of blocks, including: Matlab-blocks and Xilinx-blocks. The Matlab-blocks are generated from Matlab functions. The Matlab functions can describe only *one* clock cycle. The Xilinx-blocks are basic RTL blocks (like: registers, shifters, delays and DSPs). The design system can be easily verified within the Simulink environment. Matlab functions can be used to generate input, expected output and design parameters (like data bus width). In this way the Simulink simulation is always the same precision/configuration as the Matlab model.

In Figure 3.3 there is a small part of the filter design showed in Simulink. The blue blocks are the Xilinx-blocks. For instance, on the bottom-left there is a memory used, called 'AccumulateOut'. The address selection of the 'AccumulateOut' memory is also build with Xilinx-blocks. The 'AccumulateOut' memory is the input for the

[1]Simulink is developed by MathWorks®[12].

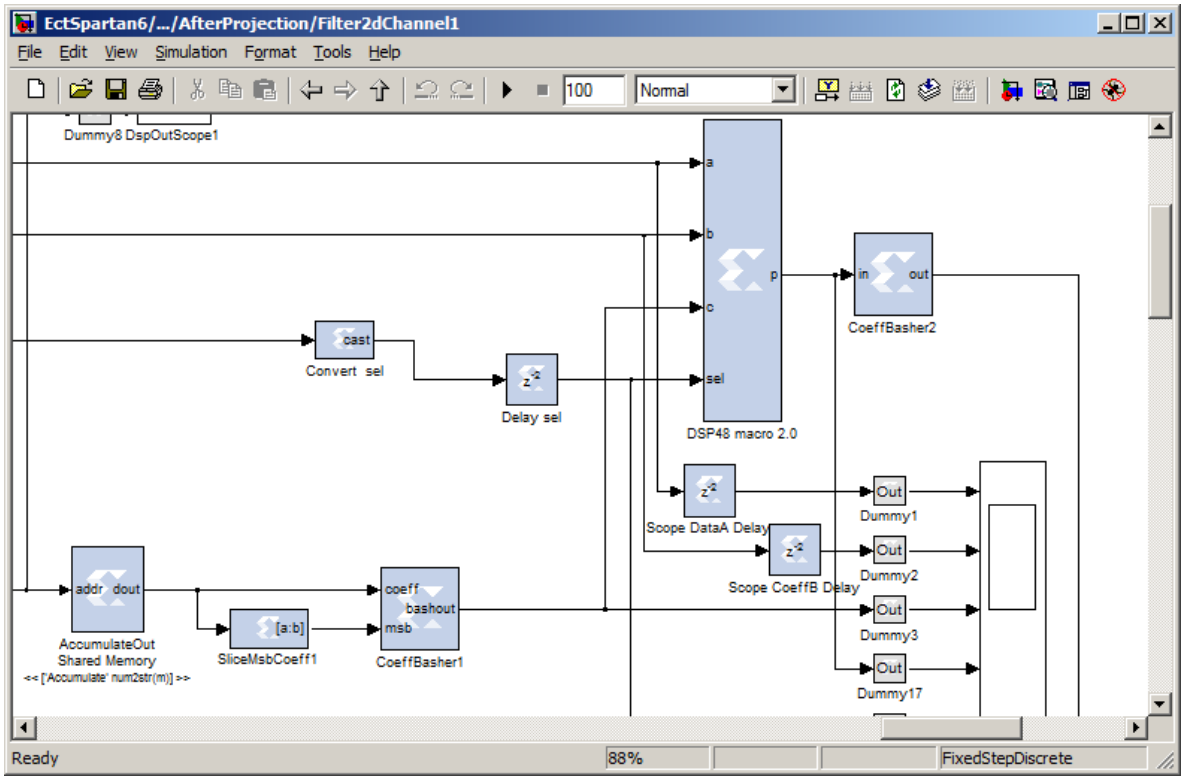


Figure 3.3: Small part of the filter design showed in Simulink.

DSP (biggest blue block) called 'DSP48 macro 2.0'. To synchronize the four inputs of the DSP, Xilinx-delay-blocks are added. Bottom-right there is a 'digital scope'(white block) added. If the simulation is started, the scope is able to plot waves/diagrams with the received data.

Lismar uses Simulink as described above, but hardly no Matlab-blocks are used. The whole system design is build up with Xilinx-blocks. In this way designing is consuming time because it requires very detailed knowledge about the system hardware implementation. The most important reason for Lismar to use Simulink is because of the debugging and validation facilities.

3.3 HLS tools

There are many different High-Level Synthesis tools to generate RTL for FPGAs. All these tools claim their own specialisms and qualities. In Figure 3.4 a number of High-Level Synthesis tools is schematically showed. For all the tools there is a valid license available. The tools are divided in two categories. The first category division is *visual* and *non-visual*. For *visual* tools it is possible to create a result by almost only using drag-and-drop within a Graphical User Interface (GUI). For *non-visual* tools the result mainly depends on a written source code. Also *non-visual* tools often include a GUI to run the compiler and to change settings. The second category division is *high-abstraction programming languages* and *domain specific languages*. If a *high-abstraction*

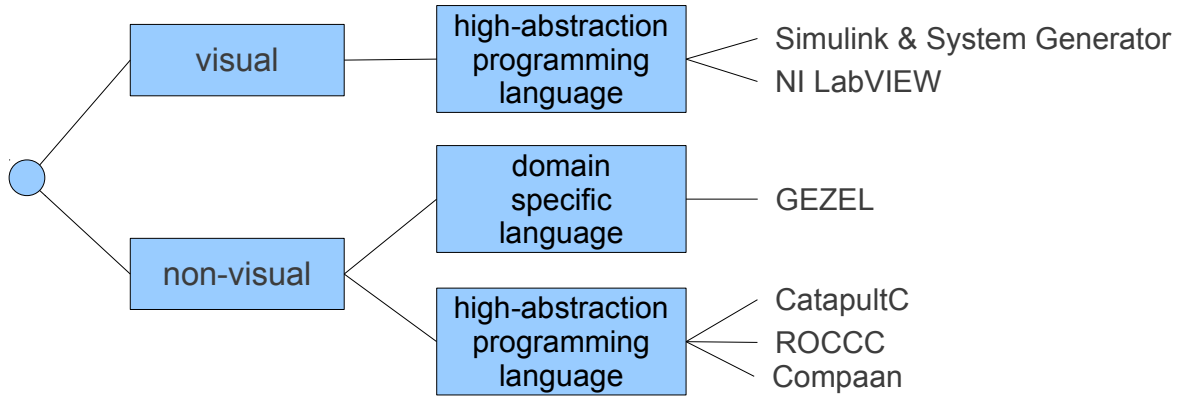


Figure 3.4: Categorized tree with tools for High-Level Synthesis.

programming language is used, the source is not created for one specific target or device. The compiler is able to generate results for several targets without changing the source. If a *domain specific language* is used, switching the target or device can not be done without changing code. The tools in the tree of Figure 3.4 are reviewed shortly in the next sections, except the software tool 'Simulink & System Generator' (see Section 3.2).

3.3.1 NI LabVIEW

LabVIEW is a well known visual programming environment of National Instruments. It can be used for many different purposes. It allows engineers and scientists to develop measurement, test, and control systems using graphical icons and wires. In the special graphic programming mode[9] it is possible to create hardware designs by drag-and-drop components and link them together. This graphical language is named 'G'. The code re-usability is high because of the high-abstraction. Several properties are available, like: Interactive Debugging, Automatic Parallelism and Performance, Combining with Other Languages. In LabVIEW many libraries are included to support numbers of devices.

3.3.2 GEZEL

GEZEL [16] is designed for domain-specific coprocessors. These domain-specific processors are often used for baseband processing, like: video coding and encryptions. GEZEL includes an own language and design environment. The design environment can be used to design, to verify and to implement. Verification can be done in a platform simulator. The platform simulator combines a hardware simulation kernel with a simulator for instruction-sets. New interfaces and coprocessors can be created in an interactive way. GEZEL claims to create good results in compare with other tools. The GEZEL language is compact and minimizes the design iteration-time.

3.3.3 CatapultC

Mentor Graphics designed a HLS tool to synthesize ANSI C++ to RTL called CatapultC[5]. CatapultC is a professional tool often used in production environments. The tool includes several key futures: *changing schedule*, *showing critical path* and *easily test benching*. The generated *schedule* from the ANSI C++ source code can be changed by setting some parameters. Manually the schedule can also be changed to met very high timing constraints. The *critical path* can be shown to find bottle necks in the design. A *test bench* can easily be generated from a separated ANSI C++ source file.

3.3.4 ROCCC 2.0

ROCCC 2.0[7] (Riverside Optimizing Compiler for Configurable Computing) is a C to HDL compilation framework. It is free and open source, designed at the University of California, Riverside. ROCCC supports a subset of the C programming language. ROCCC 2.0 does not focus on the generation of arbitrary hardware circuits. Rather, the focus is on compile time transformations and optimizations aimed at providing an application substantial speedup by replacing regions in software with a dedicated hardware component.

The Key Features of ROCCC 2.0 are:*Re-usability* and *Platform Abstraction*. *Re-usability* because of the bottom-up approach that allows the use of small modules which can be designed and automatically integrated multiple times in bigger system(s). *Platform Abstraction* makes it possible to use modules and systems on very different platforms without modifying the source code.

3.3.5 Compaan

The Compaan Design tool (Compaan)[3] is able to extract the concurrency available in the code and map it to *distributed processes* and *distributed memories*. Distributed processes can run without much interference with other processes. Distributed memories handle the exchange of data between distributed processes without using large global shared memories. The distributed processes and their connections are stored in a Kahn Process Network (KPN)[17]. The KPN is a deterministic model in which it is possible to map processes to hardware (with own frequency) or software. Every interprocess communication is done by an own First In First Out memory (FIFO). This FIFO avoids resource contention. The processes wait until new data is available in the FIFO (blocking read). Because of these FIFO's with blocking read, there is no need for interprocess synchronization and a global scheduler to manage the different processes.

3.4 Conclusion

In this chapter a list HLS tools is given. It is too much time consuming to test every tool listed. Therefore a pre-selection is made on category 'visual higher programming languages', 'non-visual domain specific languages' or 'non-visual domain specific languages'.

The first category (visual higher programming languages) contains both Simulink and LabVIEW. As told in Section 3.2 Lismar is using Simulink for there current implementation at this moment. The approach of LabView varies less with Simulink, therefore the opportunity that Lismar will increase the implementation effort with LabVIEW is small.

The second category (non-visual domain specific languages) contains GEZEL. GEZEL is designed for platforms with special calculation modules. The current Xilinx FPGA (included in the current hardware implementation) doesn't include very special calculation modules.

The last category (non-visual higher programming languages) contains tools with a high abstraction level, which is desired. The non-visibility is in line with the current development method in Matlab, so parsing the Matlab input can be straightforward. This category gives the biggest opportunity to improve the implementation.

By choosing the last category three tools remain, namely CatapultC, ROCCC and Compaan. These tools are benchmarked with a use case in the next Chapter. Depending on the score, one of the tools will be selected to get used for the final implementation of the mathematical model of the inspection system.

4 Benchmarking

In this chapter the three selected tools (CatapultC, ROCCC and Compaan) of the previous chapter are benchmarked. Benchmarking is done by implementing a use case with the three tools. In the use case the data have to be processed in real-time. Finally every used tool for implementation of the use case will be scored. The tool with the highest score is selected for the final implementation of the system.

4.1 Use case

For benchmarking the selected tools a use case is defined. The mathematical equation of the use case is defined in Equation 4.1.

$$result = \left[\left[\sum_{s=1}^{204} ADC_{(a,s)} * coefficient_{(c,s)} \right]_{c=1}^{10} \right]_{a=1}^{20} \tag{4.1}$$

The inputs of the equation are ADC and *coefficient*. The ADC contains 20 channels with each channel 204 samples. The *coefficient* contains ten tables with each 204 values. A dot product of ADC is calculated for each coefficient table. In total there are 200 final results. These results have to be outputted in serial.

There are two timing constraints. The first timing constraint is for the ADC input. The ADC input is sampled with a speed of 10MHz. The second timing constraint is for the throughput. The throughput for all the multiplications and additions has to be 20400ns.

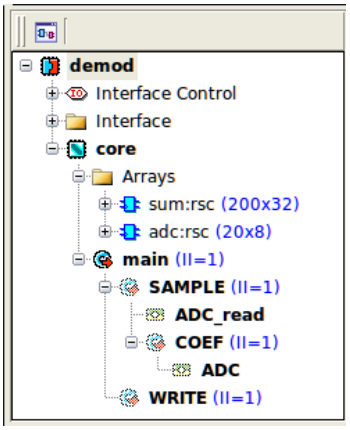
4.2 CatapultC

The first steps of the implementation of the use case in CatapultC are intuitive. After the graphical user interface of Compaan is started, a new project can be created. Creating a project can be done by selecting *source files*, specifying a *design frequency*, choosing a *target device* and selecting some *libraries*. The selected *source file* for the use case is displayed in Figure 4.1a. The difference between the C-code for CatapultC and a possible C-code for a pc application is minimal. The type changes of the inputs/output from arrays to channels(line 3-5) and the labels(line 10,13,15,18,21,26) before the loops (optional) are the main differences. The arrays are replaced by channels (FIFO's), because CatapultC has to known in which order the inputs are received. The *design frequency* of the system is set to 100MHz. The *target device* is set to the FPGA device used in the hardware implementation of Lismar . All the possible *libraries* are selected.

If the project is created and the source files are compiles, the 'architecture constraint' can be changed to met the timing constraints (see Figure 4.1b). The ADCs have to be calculated

```
1  #include <ac_channel.h>
2
3  void demod(ac_channel<short> ADC_in[20],
4             ac_channel<short> &coefficient,
5             ac_channel<int> &result)
6  {
7    int sum[20*10];
8    char coef, adc[20];
9
10   INIT:for (int a=0; a<20*10; a++)
11     sum[a] = 0;
12
13   SAMPLE:for (int s=0; s<204; s++)
14   {
15     ADC_read:for (int a=0; a<20; a++)
16       adc[a] = ADC_in[a].read();
17
18     COEF:for(int c=0; c<10; c++)
19     {
20       coef = coefficient.read();
21       ADC:for (int a=0; a<20; a++)
22         sum[a*10+c] = sum[a*10+c] + adc[a]*coef;
23     }
24   }
25
26   WRITE:for (int a=0; a<20*10; a++)
27     result.write(sum[a]);
28 }
```

(a) input: C-code



(b) constraint

Figure 4.1: CatapultC experiment

in parallel to process the data fast enough (because the system is running on 100MHz). Therefore the loops *ADC_read*(line 15) and *ADC*(line 21) are fully enrolled. All other loops are pipelined with interval 1. Every clock cycle a new value is processed by setting the pipeline to interval 1. To avoid a memory bottle neck, the shared memory *sum* is changed from memory to registers.

After the 'architecture constraint' is set the system can be scheduled. With the settings described above the system throughput is 2240 clock cycles totally. To met the timing constraints there are 200 clock cycles too much. These 200 clock cycles are used for the serialization(line 26-27) of the output. In theory this serialization can be run in parallel with the process of the next 204 samples. Unfortunately CatapultC an not be forced to do that. Perhaps it can be forced in the 'Full version' of CatapultC. For this experiment the 'University Version' is used. In the 'Full Version' it is possible to create systems with hierarchy. Maybe it is possible to create a hierarchical design that outputs the results while processing the next samples in parallel. An other option to met the timing constraints, is increasing the clock speed from 100MHz to 110MHz. By increasing the clock speed the total run time is decreased from 22400ns to 20361ns. After changing the clock speed the system meets the timing constraints.

The VHDL-code can be generated, simulated and synthesized. The simulation can be started with a single mouse click in the graphical user interface of CatapultC (if there is an additional Modelsim license). To verify the simulation results a test bench can be created in a separated C-code source file. After the simulation, the design has to be synthesized with one

of the supported compilers (the XST compiler of Xilinx is not supported, see Section A.1).

A drawback of the CatapultC tool is the absence of a list with limitations of the 'University Version'. The only message found is: "This version may only be used for academic purposes. Some optimizations are disabled, so results obtained from this version may be sub-optimal." Furthermore the annual costs of CatapultC is very high, beside the CatapultC license also a license of a supported compiler is necessary for synthesis and the Modelsim licence is recommended.

4.3 ROCCC

ROCCC is an open source compiler as explained in Section 3.3.4. Together with the ROCCC compiler a clear user manual[8] is provided. In the user manual there is a chapter about coding guidelines. The coding guidelines include a short list of limitations (see Table 4.1). For the most limitations is an easy workaround. The last two limitations are the most restricted ones.

1) Logical operators that perform short circuit evaluation. The "&" and "|" operators do work and should be used in place of "&&" and "||"
2) Generic pointers
3) Non-component functions, including C-library calls
4) Shifting by a variable amount
5) Non-for loops
6) Variables named 'C'
7) The ternary operator (?:)
8) Initialization during declaration
9) Array accesses other than those based on a constant offset from loop induction variables

Table 4.1: ROCCC 2.0 current code limitations [8, p.43].

The ROCCC compiler is supplied as plug-in for the Eclipse IDE[4]. Therefore the ROCCC compiler is easy to use (especially for those who already used Eclipse before). The plug-in adds a number of additional buttons, namely: *creating a new system*, *creating a new module*, *building a system* and *managing libraries*. ROCCC is designed to use a bottom-up approach (hierarchical design). Therefore the user can create small modules and use these modules (multiple times) in system(s). If the user starts to build system, it is possible to set some compiler parameters. For example: loop enrolment, loop fusion, inline module and parallelism.

It was hard to implement the use case in ROCCC. After spending lot of time the implementation is still not functioning correctly. In Figure 4.2 the source code of the best achieved result is shown. The outer loop(line 5) selects the different coefficient tables and the inter loop(line 8) selects every sample. The source code looks very simple, but less variation is possible.

The loop labeled as *UNROLL*(line 8) has to be fully unrolled, because the variable *currentSum* is set to zero(line 7) and the total results are saved(line 10). If this loop is not fully enrolled the compiler fails. Because the loop *UNROLL* is fully enrolled ROCCC generates an implementation with 204 multiplier in parallel. This is not a desired behavior because the samples of the ADC are available with a frequency of 10 MHz and could better be processed in serial. Exchanging the loops to get the desired implementation is not possible, because ROCCC fails to add the multiplication results together.

```
1  void demod_sys(int* in, int** coef, int *out)
2  {
3    int c, s, currentSum ;
4
5    for (c = 0; c < 10; ++c)
6    {
7      currentSum = 0 ;
8      UNROLL:for (s = 0; s<204; s++)
9        currentSum += in[s] * coef[c][s];
10     out[c] = currentSum;
11   }
12 }
```

Figure 4.2: ROCCC experiment

4.4 Compaan

Like the other tools, Compaan uses a Graphical User Interface. Generating a hardware implementation with Compaan can be done in two steps. In the first step, a KPN is created from a given source file. In the second step the KPN is mapped to a Xilinx XMP-file.

4.4.1 Creating a KPN

A KPN is created from a source file. The source file of the test case is included in Figure 4.3. Four design rules have to be applied to be able to generate a KPN.

1. **A pragma(line 5) has to be added at the top of the main function.** The pragma indicates which function must be used to create a KPN.

2. **The inputs(line 7-14) have to be provided as an array and only be used once.** The input arrays are implemented in hardware as a FIFO by Compaan, therefore the input order must be known. The input order can be specified by copying the input array to a local buffer(line 26-29) or copying it to a local variable(line 43). Both implementation methods are equal. *It is important to use the input array at only one position of the source file. If the input should be used twice the input must ne copied to a local variable.*

3. **The output(line 15) has to be provided as an array and only be used once.** The implementation of the output is like the inputs only reversed.

4. **The mathematical parts of code have to be moved to separate functions(line 1-3).** In this way initialization and data streams are divided from the computations. Compaan is not a complete solution, it only manages the data streams. For these computational functions an additional pragma is required(line 1) to indicate how the functions should be called externally (see for more information Section 5.4).

The principle of the four applied design rules are not difficult. Additionally if more parallelism is desired statements should be copied multiple times(line 26-29, 34-37,44-47). The possibility to unroll (unfold) loops is not working proper in Compaan at this moment. If the source file is ready, the KPN can be generated.

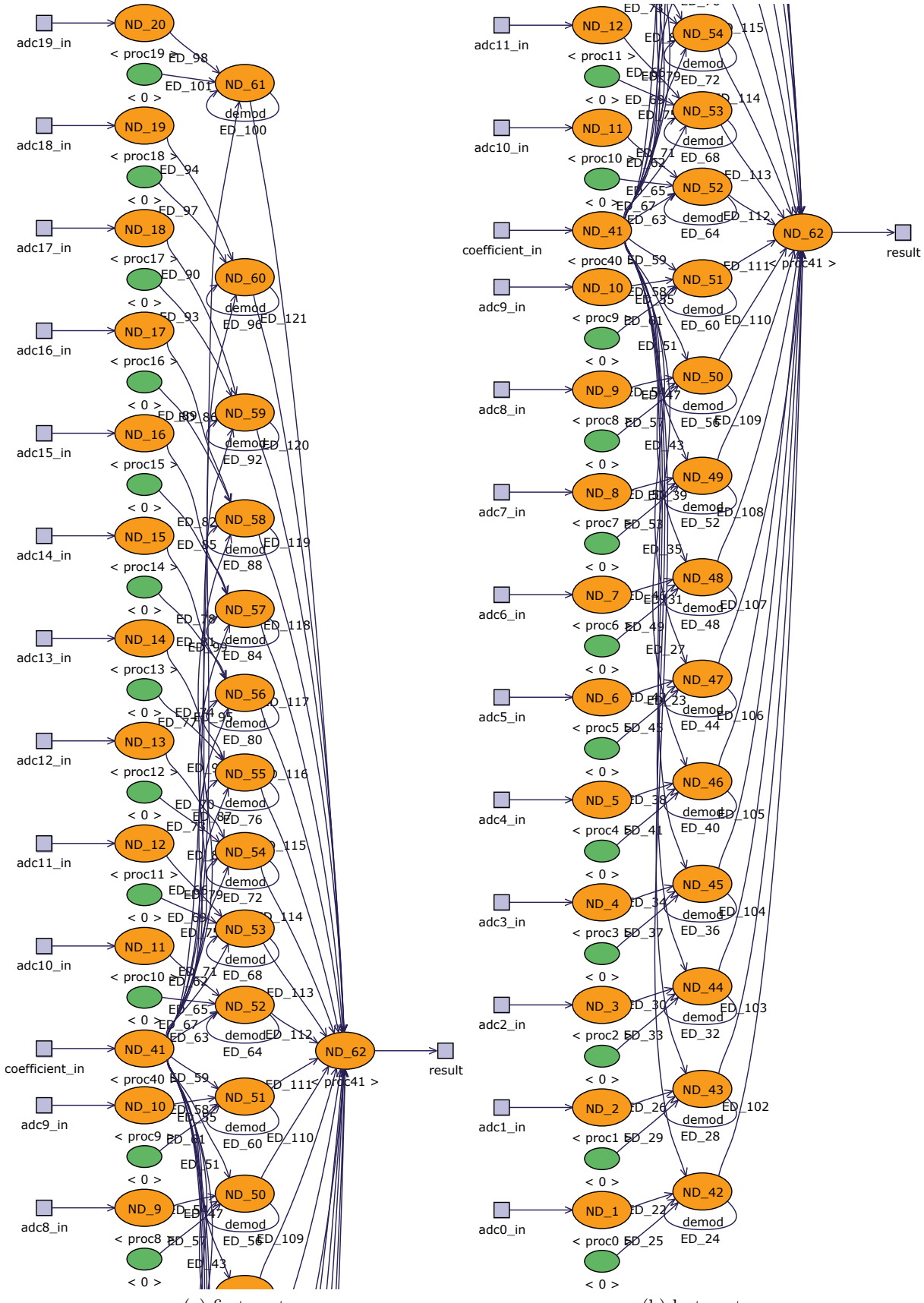
4.4.2 Mapping the KPN

If the KPN is generated it can be displayed graphically (see Figure 4.4). In this stage the designer can analyze the design and think about more parallelism. On the left of the KPN there are 21 input nodes, *adc0_in* up to *adc19_in* which are buffered in *NODE_1* up to *NODE_20* and *coefficient_in* which is buffered in *NODE_41*. In the middle *NODE_42* up to *NODE_61*, these are nodes which contain the computational function(line 1-3). These computational nodes need three inputs: a *sample*, a *coefficient* and an *intermediate result*. The *sample* is coming from one of the input buffers, the *coefficient* is coming from *NODE_41* and the intermediate result is coming from the self-loop. The intermediate results are written to the self-loop and the final result is written to *NODE_62*. *NODE_62* serializes the output of the computational functions and writes it to the output(line 50-52).

The KPN can be mapped to a Xilinx XMP-file. The XMP-file can be simulated in the Xilinx ISE and synthesized in the platform generator. In the simulations the timing of the model can be validated. The results and timing of the implemented use case are correct.

```
#pragma compaan_property pipeline 2
void demod(short sample, short coef, int sumi, int * sumo)
{ *sumo = sumi + sample * coef; }

#pragma compaan_procedure demod_20
void demod_20(
  short coefficient_in[10][204],
  short adc0_in[204],  short adc1_in[204],  short adc2_in[204],
  short adc3_in[204],  short adc4_in[204],  short adc5_in[204],
  short adc6_in[204],  short adc7_in[204],  short adc8_in[204],
  short adc9_in[204],  short adc10_in[204], short adc11_in[204],
  short adc12_in[204], short adc13_in[204], short adc14_in[204],
  short adc15_in[204], short adc16_in[204], short adc17_in[204],
  short adc18_in[204], short adc19_in[204],
  int result[20*10])
{
 int sum[20][10];
 short adc[20][204];

 //helpers
 #define adc_read(i) adc[i][s] = adc ## i ## _in[s]
 #define demod_ex(i) demod(adc[i][s], coef, sum[i][c], &(sum[i][c]))

 //read ADC
 for (int s=0; s<204; s++)
 { adc_read(0);  adc_read(1);  adc_read(2);  adc_read(3); adc_read(4);
   adc_read(5);  adc_read(6);  adc_read(7);  adc_read(8); adc_read(9);
   adc_read(10); adc_read(11); adc_read(12); adc_read(13); adc_read(14);
   adc_read(15); adc_read(16); adc_read(17); adc_read(18); adc_read(19); }

 //initialize demodulation sum buffer
 #define sum_init(adc) sum[adc][c] = 0;
 for (int c=0; c<10; c++)
 { sum_init(0);  sum_init(1);  sum_init(2);  sum_init(3);  sum_init(4);
   sum_init(5);  sum_init(6);  sum_init(7);  sum_init(8);  sum_init(9);
   sum_init(10); sum_init(11); sum_init(12); sum_init(13); sum_init(14);
   sum_init(15); sum_init(16); sum_init(17); sum_init(18); sum_init(19); }

 //execute demodulation
 for (int s=0; s<204; s++)
  for (int c=0; c<10; c++)
  {
   short coef = coefficient_in[c][s];
   demod_ex(0);  demod_ex(1);  demod_ex(2);  demod_ex(3);  demod_ex(4);
   demod_ex(5);  demod_ex(6);  demod_ex(7);  demod_ex(8);  demod_ex(9);
   demod_ex(10); demod_ex(11); demod_ex(12); demod_ex(13); demod_ex(14);
   demod_ex(15); demod_ex(16); demod_ex(17); demod_ex(18); demod_ex(19);
  }

 for (int a=0; a<20; a++)
  for (int c=0; c<10; c++)
   result[a*10+c] = sum[a][c];
}
```

Figure 4.3: Compaan experiment C-code



(a) first part

(b) last part

Figure 4.4: Compaan experiment KPN

4.5 Conclusion

	real-time	resources	TTP	messages	costs	simulation
CatapultC	+	+	+ +	+	- -	+
ROCCC	+/-	- -	- -	- -	+ +	+
Compaan	+ +	+	+	-/+	-/+	+

Table 4.2: Scores of every tool.

The reached results of the benchmark are placed in Table 4.2. In the table for every tool there are six different properties scored, namely *real-time*, *resources*, *Time To Product*, *messages*, *costs* and *simulation*. The scores per column are relative to each other. *Real-time* indicates how easy the timing constraints are met. *Resources* indicates the used hardware for the implementation. *Time To Product (TTP)* indicates the spent time to get a working result. *Messages* indicates if the error and warning messages from the compiler are clear. *Costs* indicates the license costs of the tool. *Simulation* indicates the effort to run a simulation.

Table 4.2 starts with the column *real-time*. Real-time is in our concept the most important parameter. The solution is useless if it do not meet the timing constraints. All tools meet the constraints. For CatapultC the designer has to increase the frequency. In Compaan the timing can be easily improved by duplicating function calls. The *resources* used for the implementation are almost the same for CatapultC and Compaan. The resources for ROCCC are much more. The *Time To Product (TTP)* is for ROCCC very bad, it is hard to design the code in the right way. The TTP of Compaan is in compare with CatapultC a bit longer. The compiler warning and error *messages* of CatapultC are the most clear. For ROCCC and Compaan the clearance of the messages is depending on the stage the problem is found. In later stages ROCCC almost always outputted one common error message without a line number. If the *costs* of the tools are compared CatapultC is by far the most expensive tool, the open-source ROCCC tools is for free. All tools have there own innovative way of simulation and validation, there is no big difference.

This benchmarking is done to select one tool to use for a case study. It is possible that a person who is more familiar with one of the three tools, is able to get a higher performance out of the tools. The assumption is done that the knowledge about these tools was good enough to make a fair comparison. All tools are used for at least one month before scoring the use case.

Finally, Compaan will be used because of the good average scores. CatapultC has got a better score for some properties, but the difference is small. The biggest drawback of CatapultC are the extreme high annual costs. A cost-efficient solution is for Lismar important. Compaan is not a total solution, therefore some computational nodes have to be created outside Compaan.

Implementation - CONFIDENTIAL

5

5.1

5.2

5.3

5.4

5.5

5.6

5.7

5.8 Conclusion

In this Chapter the implementation of the mathematical model is described. Four keywords (speed, optimization, hybrid and interleaving) are explained and discussed with examples from the implementation.

speed The first example shows that Compaan is able to handle data flows with critical timing constraints. Compaan is able to divide big shared memories in smaller distributed memories with parallel processes. Parallelism can easily be increased by copying lines multiple times, therefore high processing speeds can be realized.

optimization The second example shows a big memory optimization in comparison with the Lismar implementation. In the Lismar implementation data is stored redundantly, to avoid complex reordering logic. Compaan is able to generate a FIFO construction for the user to easily reorder data streams.

hybrid The third example shows how Compaan can be used in a hybrid construction. Compaan is not designed to generate the complete VHDL. The computational units should be implemented outside Compaan. Three different approaches are used to implement a computational unit, namely: manually (see Appendix B), CatapultC (see Appendix A.2) or ROCCC. In general the three different approaches score the same.

interleaving The fourth example discussed the quality of Compaan to interleave data streams. Streams can easily be interleaved and merged together. Therefore more parallelism and throughput can be achieved.

6 Results

In the previous chapter the implementation of several modules of the mathematical model are described. In this chapter the total result is evaluated and discussed. The evaluation is done by validating the output and timing performance. Finally some development 'problems' are discussed.

6.1 Output validation

To ensure a correct output of the system the total system is validated in four ways. (Most of the validations are also done for intermediate results.) For every validation the test input and output values are read from files. These files are generated with the mathematical model of the system in Matlab. In this way input and output of both systems are always the same. The input files contain 8160(=40x204) samples for each ADC, this number is equal to forty runs of the total system. The four different validations are: data flow, RTL, after synthesis and real-world.

data flow The first validation is done by compiling the Compaan source c-files with the GNU

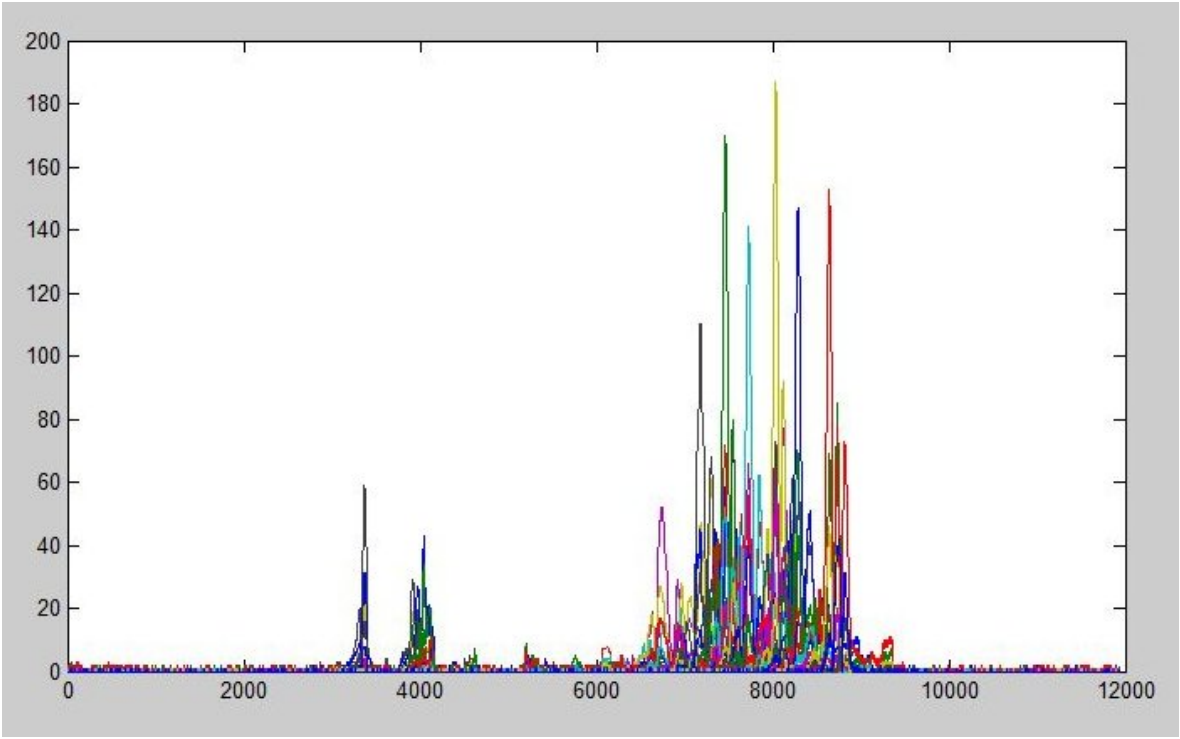


Figure 6.1: Real-world output (metal object moved over the coils).

Compiler Collection (GCC) and by running the executable with input files. To achieve this compiling two small GCC macros are written (to read the input stream and to check the output stream). This validation is the most easy and fast way of validation. This validation validates only the data flow of the system, timing and Compaan sub-blocks are not tested. This validation is done to find main problems in the data flow.

RTL The second validation can be done after mapping the KPN to an XMP-file. The generated XMP-file and the generated test bench file can be added to a Xilinx project and the RTL behavior can be simulated in Simulink. The input and output files are automatically read and processed. If the simulation results differ with output files an *ERROR* flag is set. With this simulation timing constraints can also be validated. In the test bench file a timeout value can be set to check if the whole system is ready within the specified time. But if more exact time is required, manual measurement in the wave files is necessary.

after synthesis The third validation can be done after synthesis the VHDL. Xilinx offers the possibility to simulate intermediate results in the process to generate a binary for the FPGA. The first simulation (behavioral[18]) is used to verify the RTL code and to confirm that the design is functioning as intended (this is done in the previous validation). The second simulation (post-translate[19]) verifies that design is correct after the translation process. This simulation is primarily used in this validation, but the simulations permanently failed to run. It is remarkable that basic designs do not pass this Xilinx validation, too. Xilinx confirms this problem but doesn't provide a solution. Therefore another way to validate the design after synthesis is applied. This validation is done by adding tables with static ADC samples to the design and by running this modified design on the real-platform. The output received by the network is compared with the proposed output.

real-world The fifth validation is the final validation. It is almost the same validation as the previous one. Only this time with real ADC samples. A metal object is moved over the coils and the results are interpreted for a natural result on the terminal. In Figure 6.1 the result is shown. A more detailed view of the picture shows the several coils reacting after each other.

6.2 Timing performance

The timing constraints of the system are based on the maximum rotation speed of the roll and the desired measurement precision. The data stream measured from the coils is 3.2 GBit per second and a common measurement takes at least 3 minutes. In this case more than 500 TBits of data is gathered. Therefore it is not possible to store the measurement values temporally to process it at a later moment. The implementation built with Compaan, is able to process the data real-time.

Despite the fact that the system met the given timing constraints it is interesting to know if the implementation is time optimal. To answer this question the correlation between the estimated area and the number of cycles for the demodulation of one coil is calculated and plotted in Figure 6.2. The plotted points (blue line with squares) in the graph are calculated with the Scheduling Toolbox for MATLAB[1]. If the design is scheduled with

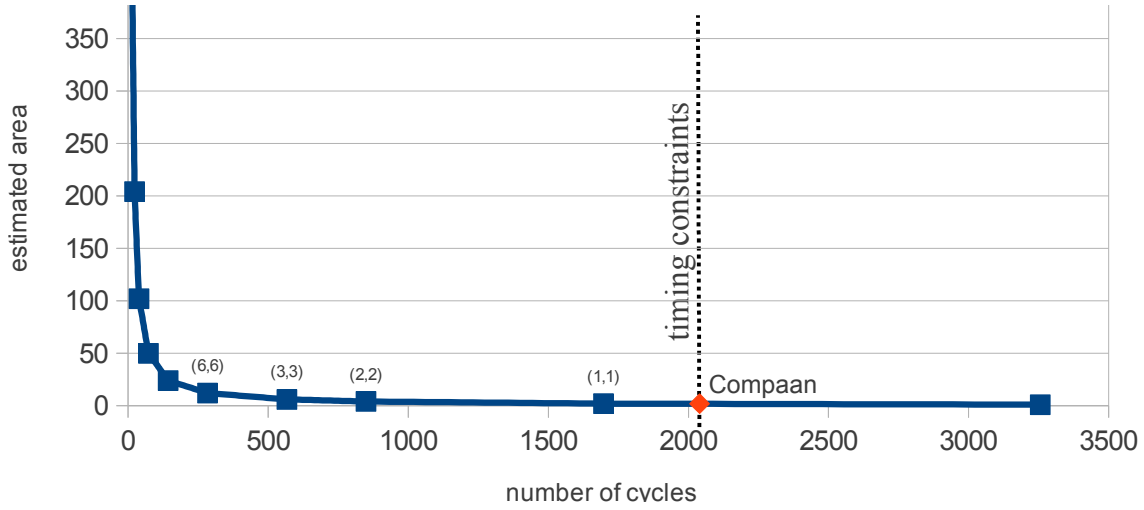


Figure 6.2: Demodulation performance for one ADC.

unlimited resources and the calculations are performed 'as soon as possible (ASAP)' nine cycles are used (asymptote on the left of the figure). The maximal number of 3256 cycles is found if all the instructions are executed in serial. The other points are calculated by limiting the available resources. Four points are marked with the used resources (number of MULTIPLIERs, number of ADDERs).

The Compaan implementation is also marked in the graph. The Compaan implementation uses 2040 cycles to complete. Within the Compaan implementation one multiplier and one adder is used. The result of the Scheduling Toolbox for one multiplier and one adder is ready in 1697 cycles. The Scheduling Toolbox calculates a better result than Compaan because it calculates the minimum number of calculations. Compaan calculates more to keep the demodulation simple. The timing of the Compaan implementation is exact within the timing constraints. If the demodulation runs faster the system will not improve. If the demodulation will run slower the system will fail. So removing the useless calculations from Compaan will not improve the quality but only increase complexity.

6.3 Comparison Simulink and Compaan implementation

As mentioned in the background (Section 3.2), Lismar implements the model in Simulink. In this section the Simulink implementation is compared against the Compaan implementation.

6.3.1 Timing

If the timing is compared, there is less difference, both implementations met the timing constraints. Therefore it is not interesting to give a detailed comparison about timing. But there is something to say about how the (timing) constraints are established. The global timing constraints are based on the maximum allowed rotation speed of the roll, two meter per second and are defined by the NDT-specialist. Based on this global constraints sub-constraints are defined, for instance the speed of the filters which is 6,8kHz. Every filter needs a coefficient table. The constraint of the coefficient table is unlike the filter timing constraint not defined by the NDT-specialist but by the hardware engineer. The hardware

engineer defines the filter coefficient size at 1000 coefficients. 1000 is the maximal number of serial instructions that can be executed within the given time (see Equation 6.1). In the equation the maximum filter and system frequency is used, therefore the filter is very area/time efficient. The Compaan design is running on 100MHz, so only 714 instructions can be executed (see Equation 6.2). To execute the same number of instructions more parallelism is necessary. Constraints optimizations like the filter coefficient size have been done more often in the system.

$$max\_instructions = \frac{\left(\frac{system_{freq}}{filter_{freq}}\right)}{input_{width}} = \frac{\left(\frac{140\text{MHz}}{7\text{kHz}}\right)}{20} = 1000 \tag{6.1}$$

$$max\_instructions = \frac{\left(\frac{system_{freq}}{filter_{freq}}\right)}{input_{width}} = \frac{\left(\frac{100\text{MHz}}{7\text{kHz}}\right)}{20} = 714 \tag{6.2}$$

6.3.2 Resources

The used resources are compared for both implementations. Two parts from the mathematical model are selected to be compared, namely: *demodulation* and one *filter* (see Figure 1.2). The demodulation is implemented only once. The filter is implemented eight times. In Table 6.1 the used resources are displayed per part. These values are obtained from the 'Module Level Utilization report' of Xilinx. Six categories are defined, namely: *Slices*, *Slice Reg*, *LUTs*, *LUTRAM*, *BRAM/FIFO* and *DSP48A1*.

	Slices		Slice Reg		LUTs		LUTRAM		BRAM/FIFO		DSP48A1	
available	23038		184304		92152		21680		268		180	
demodulation												
- Simulink	5490	23,8%	15684	8,5%	10609	11,5%	6034	27,8%	29	10,8%	33	18,3%
- Compaan	6324	27,5%	7363	4,0%	10358	11,2%	1632	7,5%	13	4,9%	29	16,1%
filter												
- Simulink	122	0,5%	206	0,1%	176	0,2%	67	0,3%	5	1,9%	1	0,6%
- Compaan	1616	7,0%	2037	1,1%	3073	3,3%	219	1,0%	15	5,6%	2	1,1%

Table 6.1: A comparison of the used resources of two parts.

The resources used for the *demodulation* implementation, differ per category. The small differences are *Slices*, *Slice Reg*, *LUTs* and *DSP48A1*. The two main differences are *LUTRAM* and *BRAM/FIFO*. More resources are used for Simulink, respectively 20% and 6%.

LUTRAM (+20%) For the Simulink implementation 20% more Look-Up Tables are used as distributed RAM(LUTRAM). This difference is introduced by the calculation of more than 200 moving averages. The Compaan implementation uses a FIFO to store the values (see Section 5.4). The Simulink implementation uses Look-Up Tables to store these values.

BRAM/FIFO (+6%) The Simulink implementation uses more BRAMs than the Compaan implementation because it stores data redundantly (see Section 5.3).

The resources used for the *filter* differ per category not more than 6,5% in compare with the total available resources. But the differences are scaled up with eight because the filter is implemented eight times. The common reason why more resources are used is explained in Section 6.3.1. Because of the used frequency of Compaan the implementation is conducted

twice and therefore it is less area efficient. The categories *Slices*(-6,5%) and *BRAM/FIFO's*(-3,7%) form the main differences.

Slices(-6,5%) The difference in Slices is because Compaan implements the filter in fifteen nodes. (Fifteen nodes are relatively many nodes.) Every node has its own switching logic.

BRAM/FIFO's(-3,7%) Bigger FIFO's and BRAM's are used to store the intermediate results in Compaan. Because Compaan can only handle FIFO's with a bit-width of 8,16,32,64. With Simulink it is possible to apply bit-with narrowing. For instance the DSP of the filter. The original bit-with of the DSP-output is 48bits. The 48bits are stored in a 64bit width FIFO. This is $(64 - 48) * 1000 = 16000$bits wasted space.

6.3.3 Power consumption

The power consumption is not a main issue in the implementation. But if the power consumption is compared, the power consumption of the Compaan implementation is 100mA less. There are some possible factors that contribute to this power reduction.

- The clock speed of the Compaan system is lower (140MHz versus 100MHz).
- The Compaan implementation uses half of the BRAMs of the Simulink implementation.
- The wrappers (used to read the BRAMs) are disabling the BRAMs when they are not used.

6.4 Development problems with Compaan

In this section some development problems achieved with Compaan are listed. Note that the implementation is done with the 'nice alpha 2011' version of Compaan. This version can be used for production purposes but is still under active development. This list of problems is already reported and probably fixed in the next release. For all the problems below there is a (suboptimal) workaround.

- **RTL level simulation of hierarchical designs is not possible.** To keep the Compaan models more readable, it is possible to create designs with hierarchy. Simulating the total hierarchical design at RTL level is not possible. It is only possible to simulate the sub-levels independent of each other.
- **Some times the KPN-rate-matcher fails.** As explained in Section 5.3, some times the KPN-rate-matcher failed to calculate the most optimal size of the FIFO's. Therefore manually changes are preferred. Note that there is a message to warn the user most of the time.
- **The absence to run code only once for initialization.** In the Compaan implementation several times a self-loop is used for node. The self-loops are used to store intermediate results, (for instance at the moving average in Section 5.4). At the initialization (only once), these self-loops have to be initialized with zeros. In Compaan it is not possible to execute code only once. Therefore a simple patch is built in the current implementation. This patch defines an initialization signal and changes the switching logic of the node a little bit.

- **Only restricted bit-widths possible.** The size of the FIFO's within Compaan are limited to a bit-width of: 8, 16, 32 or 64. Often these widths are not optimal sizes and so resources are wasted for too big FIFO's (see Section 6.3.2).
- **More nodes are used than strictly necessarily.** The final implementation calculated by Compaan is sometimes suboptimal. Some nodes can be removed without breaking down the system (see Section 5.3). Note that Compaan already designed an algorithm to find these useless nodes. In a future release the nodes will be removed.
- **No loop unrolling parameter.** Especially in the *demodulation* there is much parallelism. In Compaan there is no parameter to unroll loops, therefore some lines of code are duplicated twenty-four times.

Conclusion / Recommendations

7

During this thesis the mathematical model of the Lismar inspection system is implemented in hardware. In preparation of this implementation three High-Level Synthesis (HSL) tools are benchmarked. Before benchmarking these tools were explored during one month. A short summery of the tools below:

CatpultC is a professional and versatile HLS tool. This non-visual HLS tool uses ANSI C as input. The CatapultC compiler is highly configurable and the Graphical User Interface (GUI) is easy to use. More parallelism can be introduced by unrolling loops. To be able to create parallelism, shared memory should be avoid in the ANSI C file (or mapped to registers). The high annual cost is the main drawback of this tool.

ROCCC 2.0 is an open source HLS tool which accepts a subset of the C-programming language and is freely available. The small list of coding limitations (included in the manual) makes it hard to design a complex system. Many skills are necessary to create bigger systems and models in ROCCC. ROCCC is still under development.

Compaan is a powerful tool designed to handle data streams with high timing constraints. Like the other tools the input of Compaan is an ANSI C file. The power of Compaan is the capability to change shared memories (used in the source code) into distributed memories with concurrent processes. Compaan generates not a full solution. Computational nodes have to be created outside Compaan.

After the benchmark Compaan is chosen to implement the mathematical model. The benchmarking shows that Compaan is able to implement complex systems in a short time. More parallelism can be created by duplicating the function calls of the computational units. Shared memories are automatically split into distributed memories. Distributed memories provide the input data of the computational units without interference. The learning curve to use Compaan is steep, the designer should learn to think in the Compaan approach of 'nodes' and 'edges'.

In Compaan the implementation of the mathematical model is done in four months. This is approximately four times faster than the implementation with Simulink. Also future maintenance of the Compaan implementation will be faster because the Compaan implementation is of a higher abstraction. Critical analyze of the Simulink implementation confirms that detailed knowledge about the system is required to design and maintain the Simulink implementation. For instance: some implementation related questions were hard to answer for the hardware engineer even a few months after implementation.

Using Compaan in a hybrid construction will give efficient results. A second tool to generated the computation nodes is desirable. Although a hardware engineer with average VHDL scales will be able to create most of the computational nodes manually by using the

pipeline templates. The user should keep the computational nodes as simple as possible, by moving logic to Compaan as much as possible.

The resources used for the whole implementation in Compaan are approximately 15% higher than the resources used in the Simulink implementation. Compaan is under active development and improves. Although Compaan is still in an alpha version the tool is very useful. Using Compaan is recommended for the implementation of systems with a complex data-flow in both small and medium-sized enterprises.

7.1 future work

Many design optimizations related to more parallelism can be easily applied by using Compaan. Below a list of optimisations. (Note if these optimisations are applied with Simulink implementation would take much more time.)

- Increase sample speed to handle higher rotation speed of the roll.
- Use more coils to increase scan speed.
- Increase filter coefficients to improve filtering.
- Lower clock speed to decrease power.

Finally, the interfacing between the already designed network of Lismar and the Compaan system gave the most problems during the first implementation state of the system. However Compaan provides board templates. In these board templates the network interface is already designed. In future designs such a board template could be very useful.

Bibliography

[1] Ing. H.J. Lincklaen Arrins, *Scheduling Toolbox for MATLAB,* *http://ens.ewi.tudelft.nl/Education/courses/et4054/Lab2011/msclab2008_RG.pdf*, 2008.

[2] Semiconductor Industry Association., *The International Technology Roadmap for Semiconductors*, (2009), 10.

[3] Compaan Design, *heterogeneous compilation*, http://www.compaandesign.com/.

[4] Eclipse, *IDE for C/C++ Developers*, http://www.eclipse.org/downloads.

[5] Mentor Graphics, *Catapult C Synthesis*, http://www.mentor.com/esl/catapult/.

[6] _____, *Communities*, http://communities.mentor.com/.

[7] Jacquard Computing inc, *Riverside Optimizing Compiler for Configurable Computing*, http://www.jacquardcomputing.com/roccc/.

[8] _____, *ROCCC 2.0 User's Manual - Revision 0.6*, feb. 2011.

[9] National Instruments, *The Benefits of Programming Graphically in NI LabVIEW*, http://www.ni.com/labview/whatis/graphical-programming/.

[10] A. Madariaga, J. Jime andnez, J.L. Marti andn, U. Bidarte, and A. Zuloaga, *Review of electronic design automation tools for high-level synthesis*, Applied Electronics (AE), 2010 International Conference on, sept. 2010, pp. 1–6.

[11] MathWorks®, *MATLAB - The Language Of Technical Computing*, http://www.mathworks.nl/products/matlab/.

[12] _____, *Simulink - Simulation and Model-Based Design*, http://www.mathworks.nl/products/simulink.

[13] _____, *Using Simulink with Xilinx System Generator for DSP*, http://www.mathworks.nl/fpga-design/simulink-with-xilinx-system-generator-for-dsp.html.

[14] Gordon E. Moore, *Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff.*, Solid-State Circuits Newsletter, IEEE **20** (2006), no. 3, 33 –35.

[15] NDT resource center, *Introduction to Eddy Current Testing*, http://www.ndt-ed.org/EducationResources/CommunityCollege/EddyCurrents/cc_ec_index.htm.

[16] Patrick Schaumont, Doris Ching, and Verbauwhede Ingrid, *An interactive codesign environment for domain-specific coprocessors*, jan. 2006.

[17] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprette, *System design using khan process networks: the compaan/laura approach*, Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings, vol. 1, feb. 2004, pp. 340 – 345 Vol.1.

[18] Xilinx®, *Performing Behavioral Simulation*, http://www.xilinx.com/itp/xilinx10/isehelp/pp_p_process_simulate_behavioral_model.htm.

[19] ———, *Performing Post-Translate Simulation*, http://www.xilinx.com/itp/xilinx10/isehelp/pp_p_process_simulate_post_translate_model.htm, 2008.

[20] Xilinx®, *FPGAs*, http://www.xilinx.com/products/silicon-devices/fpga.

CatapultC design notes

A

This appendix contains two design notes for the use of CatapultC. The first note describes the problem of synthesizing CatapultC code with the Xilinx XST synthesizer. The second note describes how a Compaan computational node can be created with CatapultC.

A.1 Xilinx XST synthesis tool

No official messages can be found about the CatapultC compiler support. At the start of the project several times the Xilinx XST synthesizer was used to compile the CatapultC results. It was very confusing that the results didn't work properly. Searching on internet the Mentor Community[6] was found. Posts of the mentor community mentioned several times that the Xilinx XST synthesizer is not supported.

In practise simple models can be synthesized without warnings and errors. But the synthesized models are not correct, sometimes. For example: Block RAMs are implemented with LUTs and the timing constraints do not meet.

A.2 Create node for Compaan

Compaan supports the possibility to link CatapultC computational nodes. If the correct steps are done, linking is very easy and exists of four steps.

First the CatapultC module should be referenced in the Compaan C-code. In Listing A.1 there is an example of the computational node MAC with three inputs (*sample*, *coef*, *sum*) and one output (*result*). The first pragma(line 1) in the listing specifies that the function is implemented in CatapultC. The second pragma(line 2) is optional and specifies that the function is provided as EDIF-file[1]. It is also possible to remove this pragma and provide a VHDL-file, but this is not recommended (for more details see Section A.1). Note that it is not strictly necessary to specify the function contents in Compaan, but it can be used for C-code simulations and it increases the readability of the code.

```
1 #pragma compaan_property mapto catapultc   /* first pragma */
2 #pragma compaan_property netlist edf       /* second pragma (optional) */
3 void MAC(short sample, short coef, int sum, int *result)
4 { *result = sum + (sample * coef); }        /* not strictly necessary */
```

Listing A.1: CatapultC reverence in Compaan

[1]EDIF (Electronic Design Interchange Format) is a vendor-neutral format in which Electronic netlists and schematics are stored

Secondly a new CatapultC project must be created. It is very important to set a number of global settings. Open the settings window by menu 'Tools'→'Set Options...' and apply the values listed in Listing A.2. Add a C-file to the project with the content of Listing A.3. Go to 'Architecture Constraint' and change the 'Resource Types' of the input interfaces to 'mgc_ioport.mgc_in_wire_wait' and the 'Resource Types' of the output interfaces to 'mgc_ioport.mgc_out_buf_wait'. Thirdly generate the RTL and synthesize the design with Precision.

```
Output
    [v] VHDL
    [v] Package Output in Solution dir
Flows
  Precision RTL
    [ ] Add IO Pads
```

Listing A.2: CatapultC Options

```
1 #include <ac_channel.h>
2 void MAC(ac_channel<short> &sample, ac_channel<short> &coef,
3           ac_channel<int> &sum,     ac_channel<int> &result)
4 {
5   result.write(sum.read() + sample.read() * coef.read());
6 }
```

Listing A.3: CatapultC computational node

Finally copy the EDIF-file (e.g. psr_vhdl_impl/MAC.edf) to Compaan (e.g. projectname_KpnMapper_1/pcores/Functions_v1_00_a/netlist/MAC.edf). Note if the second pragma(line 2) in Compaan is removed then copy the VHDL-file (e.g. concat_MAC.vhdl) to Compaan (e.g. projectname_KpnMapper_1/pcores/Functions_v1_00_a/hdl/vhdl/MAC.vhd).

Compaan pipeline

B

Compaan is able to generate a pipeline template. Two parts of code must be added to generate a template(see Figure B.1). First a pragma(line 1) must be added and after that a C-function description(line 4-6). The generated pipeline template is displayed below Figure B.1.

```
1 #pragma compaan_property pipeline 3
2 void combi(int intermidate_in,  int sum_in,
3            int sum_last10_in,   int offset_gain_in,
4            int *sum_last10_out, int *intermidate_out, int *ans_out) { }
```

Figure B.1: Source to generate a pipeline template in Compaan.

```
1  -- File automatically generated by KpnMapper
2  -- This file defines a template for pipelined function implementation
3  -- Function "projection_combi"
4
5  library ieee;
6  use ieee.std_logic_1164.all;
7  use ieee.numeric_std.all;
8
9  library common_v1_00_a;
10 use common_v1_00_a.hw_node_pkg.all;
11
12 entity projection_combi_pipeline is
13    generic (
14       c_STAGES    : natural := 1;
15       N_CNTRS     : natural := 1;
16       CNTR_QUANT  : natural := 32;
17       CNTR_WIDTH  : t_counter_width := ( 0=>10, 1=>10, 2=>9, others=>10 )
18    );
19    port (
20       RST    : in  std_logic;
21       CLK    : in  std_logic;
22       -- Inputs
23       ip_intermidate_in   : in  std_logic_vector(31 downto 0);
24       ip_sum_in   : in  std_logic_vector(31 downto 0);
25       ip_sum_last10_in   : in  std_logic_vector(31 downto 0);
26       ip_offset_gain_in   : in  std_logic_vector(31 downto 0);
27       -- Iterators
28       it_index : in  std_logic_vector(CNTR_WIDTH(0)-1 downto 0);
29       it_set : in  std_logic_vector(CNTR_WIDTH(1)-1 downto 0);
30       it_i : in  std_logic_vector(CNTR_WIDTH(2)-1 downto 0);
31       -- Outputs
32       op_sum_last10_out : out std_logic_vector(31 downto 0);
33       op_intermidate_out : out std_logic_vector(31 downto 0);
34       op_ans_out : out std_logic_vector(31 downto 0);
35       --
36       ENi    : in  std_logic;
37       EN     : in  std_logic_vector(c_STAGES-1 downto 0);
38       STALL_FRONT   : out std_logic_vector(c_STAGES-1 downto 0);
39       STALL_BACK    : out std_logic_vector(c_STAGES-1 downto 0);
40       ERROR : out std_logic
```

```
41    );
42 end projection_combi_pipeline;
43
44 architecture RTL of projection_combi_pipeline is
45     constant error_int : integer := -1;
46     constant reset_int : std_logic_vector(0 downto 0) := b"0";
47     -- Input registers
48     signal ipr_intermidate_in : std_logic_vector(31 downto 0);
49     signal ipr_sum_in : std_logic_vector(31 downto 0);
50     signal ipr_sum_last10_in : std_logic_vector(31 downto 0);
51     signal ipr_offset_gain_in : std_logic_vector(31 downto 0);
52     -- Iterator registers
53     signal itr_index : std_logic_vector(CNTR_WIDTH(0)-1 downto 0);
54     signal itr_set : std_logic_vector(CNTR_WIDTH(1)-1 downto 0);
55     signal itr_i : std_logic_vector(CNTR_WIDTH(2)-1 downto 0);
56     --
57     -- Your pipeline signals
58     -- STAGE_0
59     signal s0_intermidate_in : std_logic_vector(31 downto 0);
60     signal r0_intermidate_in : std_logic_vector(31 downto 0);
61     signal s0_sum_in : std_logic_vector(31 downto 0);
62     signal r0_sum_in : std_logic_vector(31 downto 0);
63     signal s0_sum_last10_in : std_logic_vector(31 downto 0);
64     signal r0_sum_last10_in : std_logic_vector(31 downto 0);
65     signal s0_offset_gain_in : std_logic_vector(31 downto 0);
66     signal r0_offset_gain_in : std_logic_vector(31 downto 0);
67     -- STAGE_1
68     signal s1_intermidate_in : std_logic_vector(31 downto 0);
69     signal r1_intermidate_in : std_logic_vector(31 downto 0);
70     signal s1_sum_in : std_logic_vector(31 downto 0);
71     signal r1_sum_in : std_logic_vector(31 downto 0);
72     signal s1_sum_last10_in : std_logic_vector(31 downto 0);
73     signal r1_sum_last10_in : std_logic_vector(31 downto 0);
74     signal s1_offset_gain_in : std_logic_vector(31 downto 0);
75     signal r1_offset_gain_in : std_logic_vector(31 downto 0);
76     -- STAGE_2
77     signal s2_intermidate_in : std_logic_vector(31 downto 0);
78     signal r2_intermidate_in : std_logic_vector(31 downto 0);
79     signal s2_sum_in : std_logic_vector(31 downto 0);
80     signal r2_sum_in : std_logic_vector(31 downto 0);
81     signal s2_sum_last10_in : std_logic_vector(31 downto 0);
82     signal r2_sum_last10_in : std_logic_vector(31 downto 0);
83     signal s2_offset_gain_in : std_logic_vector(31 downto 0);
84     signal r2_offset_gain_in : std_logic_vector(31 downto 0);
85 begin
86     PIPE_REGS : process(CLK)
87     begin
88         if rising_edge(CLK) then
89             if (RST='1') then
90                 -- Something to reset?
91             else
92                 if( ENi = '1' ) then
93                     -- Input Registers
94                     ipr_intermidate_in <= ip_intermidate_in;
95                     ipr_sum_in <= ip_sum_in;
96                     ipr_sum_last10_in <= ip_sum_last10_in;
97                     ipr_offset_gain_in <= ip_offset_gain_in;
98                     -- Iterator Registers
99                     itr_index <= it_index;
100                    itr_set <= it_set;
101                    itr_i <= it_i;
102                end if;
103                -- Pipeline Depth: 3 stages
104                -- STAGE_0
105                if( EN(0) = '1' ) then
106                    r0_intermidate_in <= s0_intermidate_in;
```

```
107                     r0_sum_in <= s0_sum_in;
108                     r0_sum_last10_in <= s0_sum_last10_in;
109                     r0_offset_gain_in <= s0_offset_gain_in;
110                 end if;
111                 -- STAGE_1
112                 if( EN(1) = '1' ) then
113                     r1_intermidate_in <= s1_intermidate_in;
114                     r1_sum_in <= s1_sum_in;
115                     r1_sum_last10_in <= s1_sum_last10_in;
116                     r1_offset_gain_in <= s1_offset_gain_in;
117                 end if;
118                 -- STAGE_2
119                 if( EN(2) = '1' ) then
120                     r2_intermidate_in <= s2_intermidate_in;
121                     r2_sum_in <= s2_sum_in;
122                     r2_sum_last10_in <= s2_sum_last10_in;
123                     r2_offset_gain_in <= s2_offset_gain_in;
124                 end if;
125             end if;
126         end if;
127     end process;    -- PIPE_REGS
128     -- Output
129     op_sum_last10_out <= STD_LOGIC_VECTOR(RESIZE(UNSIGNED(r2_intermidate_in), ↩
            op_sum_last10_out'Length));
130     op_intermidate_out <= STD_LOGIC_VECTOR(RESIZE(UNSIGNED(r2_sum_in), ↩
            op_intermidate_out'Length));
131     op_ans_out <= STD_LOGIC_VECTOR(RESIZE(UNSIGNED(r2_sum_last10_in), op_ans_out'↩
            Length));
132     -- PIPE_COMB:
133     s0_intermidate_in <= ipr_intermidate_in;
134     s0_sum_in <= ipr_sum_in;
135     s0_sum_last10_in <= ipr_sum_last10_in;
136     s0_offset_gain_in <= ipr_offset_gain_in;
137     s1_intermidate_in <= r0_intermidate_in;
138     s1_sum_in <= r0_sum_in;
139     s1_sum_last10_in <= r0_sum_last10_in;
140     s1_offset_gain_in <= r0_offset_gain_in;
141     s2_intermidate_in <= r1_intermidate_in;
142     s2_sum_in <= r1_sum_in;
143     s2_sum_last10_in <= r1_sum_last10_in;
144     s2_offset_gain_in <= r1_offset_gain_in;
145     --
146     STALL_FRONT  <= (others=>'0');
147     STALL_BACK   <= (others=>'0');
148     ERROR <= '0';
149 end RTL;
```

Eddy Current response graphs - CONFIDENTIAL

C

C.1