University of Twente

EEMCS / Electrical Engineering
Control Engineering

On FPGAs with embedded processor cores for application in robotics

Roderick Colenbrander

MSc report

Supervisors:
prof.dr.ir. J. van Amerongen
dr.ir. J.F. Broenink
ir. M.M. Bezemer
ir. E. Molenkamp

August 2009

Report nr. 015CE2009
Control Engineering
EE-Math-CS
University of Twente
P.O.Box 217
7500 AE Enschede
The Netherlands

Glossary

ACE - Advanced Configuration Environment
APU - Auxiliary Processor Unit
BIT - FPGA bitstream programming file
BSP - Board Support Package
CE - Control Engineering
CW - Configware
EDK - Platform Studio Embedded Development Kit
ELF - Executable and Linkable Format
FPGA - Field Programmable Gate Array
HDL - Hardware Description Language
LVDS - Low Voltage Differential Signalling
LUT - Lookup Table
MHS - Microprocessor Hardware Specification
PLB - Processor Local Bus
SysACE - System Advanced Configuration Environment
SysGen - System Generator for DSP
UCF - User Constraints File
XMP - Xilinx Microprocessor Project

Summary

The industry makes heavy use of embedded systems. Traditionally they consist of an embedded computer running real-time software on a real-time operating system. The complexity of embedded systems has increased due to higher demands. More design effort is required to ensure that real-time requirements can be met.

The emergence of powerful, low-cost FPGAs has initiated a trend in which real-time tasks are moved from software to FPGAs. When moving tasks away from the CPU, the demands on both the CPU and software become less harsh which simplifies design. In the end a simple CPU running non-real-time software might suffice.

The Control Engineering group has acquired an FPGA platform which they intend to use for application in robotics. This platform is the Xilinx ML510 FPGA board which contains a Virtex-5 FX130T FPGA. The Virtex-5 FX130T contains a large number of logic cells and two PowerPC CPUs which allow a tight coupling between FPGA logic and software. The Virtex-5 FX130T offers a large amount of parallel computation power which is enough for performing the PID calculations of 50,000 Production Cell setups.

The main goal of this project is to investigate the use of the ML510 platform for application in robotics. The assignment consists of two parts. The first part of this assignment is devoted to the creation of a development platform. The required work required consisted of: porting Linux to the ML510, development of a ML510 configware package in order to boot Linux and setting up the FPGA and software development tools. In the second part of this assignment a robotic demonstrator is developed in which the position of a motor is controlled by the Virtex-5.

The CE design flow has been adopted with a Xilinx hardware/software co-design flow. This extended design flow, advocates to use a mix of software and configware for a controller design. This way a better design can be obtained at a lower design effort because each part can be implemented using the best technology, either software or configware, for the task.

The extended CE design flow has been successfully applied on the creation of a robotic demonstrator. This proves that the ML510 is suitable for robotics but the large amount of computation power offered by the Virtex-5 is overkill for use in simple control projects.

At this point the ML510 platform is only accessible to designers with a background in computer engineering and it can only be used in applications with low requirements on I/O performance. Here three recommendations are given for using the board and make the board more usable. First of all the ML510 should be used in complex control applications like vision-in-the-loop which can take advantage of the parallel computation power of the Virtex-5. Second, a PID controller core can be designed of which the PID parameters and sample frequency are programmable from software, this way rapid prototyping as offered by 20-sim/4C is possible. Third, a new ML510 I/O board should be developed which fixes the performance limitations of the current board.

Preface

This report marks the end of my study Electrical Engineering at the University of Twente. During the six years of my study here I have learned a lot on the academic level but also extracurriculary. The activity I enjoyed most and will never forgot was organizing of Study Project Ohiariha to Canada in 2006-2007.

I would like to thank a number of people who have assisted me during this project. First of my supervisors Jan Broenink, Maarten Bezemer and Bert Molenkamp for their support and guidance throughout the project. Second, I would like to thank Marcel Groothuis for his assistance with adding ML510 support to Embryo, discussion on the Linux kernel and help with other parts of my assignment.

As part of this assignment Linux was ported to the ML510. During this work I have been assisted by PowerPC Linux developers. I would like to thank Benjamin Herrenschmidt (IBM Corporation) and Grant Likely (SecretLab) for the dozens of hours of support they gave me by IRC and e-mail.

Finally I would like to thank my fellow students at the Control Engineering Lab and my housemates for their questions and feedback on my project for bringing it to a higher level.

Roderick Colenbrander
Enschede, August 2009

Contents

1 Introduction

1.1 Context

The industry makes heavy use of embedded systems. Traditionally they consist of an embedded computer running real-time software on a real-time operating system. The complexity of embedded systems has increased due to increased requirements. More design effort is required to ensure that real-time requirements can be met.

The emergence of powerful, low-cost FPGAs has initiated a trend in which real-time tasks are moved from software to FPGAs. When moving tasks away from the CPU, the demands on both the CPU and software become less harsh which simplifies design. In the end a simple CPU running non-real-time software might suffice.

FPGAs offer a flexible platform for building systems-on-a-chip. Traditionally FPGAs offered only a small number of programmable LUTs and a large amount of I/O pins. Modern FPGAs contain a large amount of LUTs but next to that they also integrate CPU cores, DSP blocks, memory and other cores. Thus they allow the integration of all hardware and software, of which an embedded system is composed, into a single device.

New promising applications of FPGAs have arisen, for instance, in the field of robotics. A popular research topic in robotics is the creation of human-like robots with vision. The image processing algorithms needed for vision require a large amount of computation power. Currently, these vision algorithms run on high performance general purpose processors, but these are not well suited for this task. Trade-offs between the complexity of the image processing algorithm and its execution time are needed in order to meet real-time demands. The parallel computation power of FPGAs is suitable to accelerate these tasks and allows for the creation of more sophisticated algorithms.

Development for an FPGA is a trade-off between development time, FPGA resource usage (LUTs), timing, accuracy and performance. An example of these trade-offs can be seen in the implementation of the controllers of the Production Cell setup into an FPGA (Sassen, 2009; Groothuis et al., 2008). The original software implementation made use of double precision floating-point. Due to resource constraints in the number of FPGA LUTs extra work was required in order to fit the design into the FPGA. This rework included trade-offs in accuracy by converting double precision floating-point computations to single procision floating-point or trade-offs in timing by sequential execution of floating-point calculations.

In order to be able to perform FPGA trade-off studies and to build more complex robots, a platform featuring an FPGA with a large amount of computation power has been acquired. This platform is the Xilinx ML510 FPGA board (Xilinx, 2008c) which contains a Virtex-5 FX130T FPGA. The Virtex-5 FX130T contains a large number of LUTs and two PowerPC CPUs which allow a tight coupling between FPGA logic and software. This makes it possible to realize a complete embedded control system as a system-on-a-chip.

The ML510 is a new type of platform for the Control Engineering group. From a technical point of view the platform looks promising for use in robotics. The problem is that it is not known what the strong/weak points of the platform are: how easily the potential of the platform can be revealed and whether the platform can easily be fitted into the design methodology used within the group. This assignment tries to find an answer to these questions.

1.2 Goals and approach of the project

The main goal for this assignment is to investigate the use of the ML510 in robotics. This investigation should result in recommendations on situations in which to use the ML510 and how to use the it.

In order to find these answers, three tasks are performed:

1 The creation of a development platform consisting of FPGA and software development tools.
2 Integration of the development platform into the CE design flow.
3 Development of a robotic demonstrator in which a motor is controlled by the FPGA.

The first task focuses on the creation of a development platform, so that others can easily take advantage of the ML510 platform. The second task integrates FPGA and software development tools for ML510 development into the CE design flow. As part of the third task, the modified CE design flow is applied to the creation of a robotic demonstrator which proves the usability of the ML510 for robotics. The robotic demonstrator consists of a rotating platfrom of which the position is controlled by the FPGA by actuating a motor.

1.3 CE methodology in context of this board

The process of designing a system can be described using the pyramid shown in Figure 1.1. The figure splits the design process in several stages. The design process starts at the top and by making choices a lower stage is reached and more detail is achieved. The choices made during the design result in different final products.

The pyramid model is useful for FPGA design because trade-offs need to be made during design in e.g. accuracy, development time, FPGA resource usage, timing and performance. The large capacity of the Virtex-5 allows for this sort of trade-off studies.

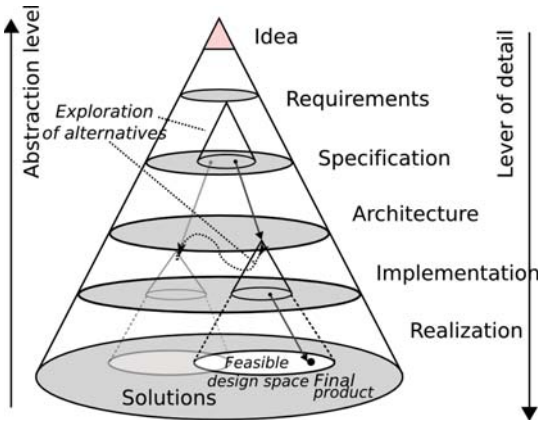

Figure 1.1: System design process (Corporaal, 2006; Groothuis and Broenink, 2009)

In case of Embedded Control Software the pyramid can be translated into Figure 1.2.

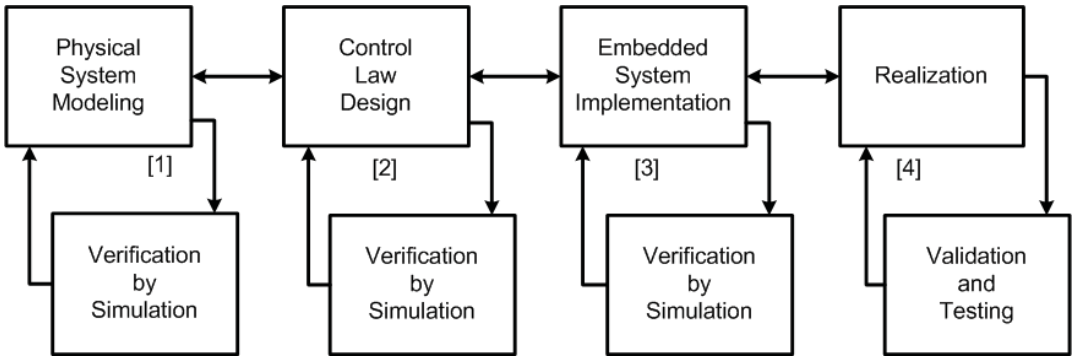

Figure 1.2: CE design flow (Broenink et al., 2007)

The figure shows a design flow developed within the Control Engineering group (CE). The design flow consists of four design phases:

- Physical System Modeling: A model of the physical system is designed.
- Control Law Design: The model of the physical system is used to create a controller for controlling the physical system.
- Embedded System Implementation: In this step the model is prepared for implementation on a hardware platform. Effects of operations like sampling, digital-to-analog / analog-to-digital conversion are included into the model.
- Realization: The system is transferred to hardware and tested on the physical system.

The result of each design phase is analyzed. In the first three design phases this is done by simulation. By simulation, it can be *verified* whether the model, as it has been written down in the simulation program, behaves as it was designed to do so. For this check, test simulations can be run to judge the behavior of the model under particular circumstances, like equilibrium, step response and transient response. Important aspects to look at are the shape of the curves and the value range of the variables. During the realization phase measurements on the real physical system can be performed. These measurement results can be compared to the simulations results, this is called *validation*.

The verification step is critical as it allows for stepwise refinement of the model. This means that a problem is divided into small, manageable subproblems. Solving of such a subproblem leads to a more complete answer to the whole problem e.g. the design of a model.

The prefered design for mechatronic design within CE is 20-sim (Controllab Products, 2009). It can be used during all four design phases from modelling a physical system to C/C++ code generation when the target hardware platform uses a general purpose processor. When the code has to be executed on an FPGA, the 20-sim code generator cannot be used. This is because FPGAs are not programmed using C/C++ but using a *Hardware Description Language* (HDL) and 20-sim does not provide a code generator for this.

In previous projects targetting an FPGA, the design was ported to Handel-C during the Realization phase. Handel-C is a hardware desciption language based on ANSI-C with language extensions for paralellism inspired by CSP. It would make sense to use Handel-C for this project but it proved not to be suitable:

- The Handel-C tools do not provide glue logic for interfacing Handel-C designs with a PowerPC CPU; writing your own Handel-C/VHDL interfaces for this is possible but it is a lot of work.
- Handel-C offers limited simulation capabilities which led to the use of trial-and-error

in previous CE projects. This method is not suitable for a large FPGA due to long synthesize times.
- The future of Handel-C is uncertain after the company which developed it (Agility Design Solutions, 2009), has been acquired by its competitor Mentor Graphics.

This project focusses on the "Embedded Control System Implementation" and "Realization" phases. Models of a plant and controller are reused from previous projects and are modified were needed. FPGA design tools from Xilinx will be used to realize the design in the Virtex-5.

1.4 FPGA terminology

In electrical engineering and computer science some terms have a different meaning. One such term is hardware which in case of electrical engineering refers to components like resistors or circuit boards while in case of computer science the content used to configure an FPGA is considered hardware as well. To prevent confusion within CE, terminology is introduced here which is used throughout the rest of this report. The terminology is introduced by explaining CPUs and software first.

A CPU is a device which can execute a fixed number of operations like additions and multiplications. The tasks which a CPU has to perform are contained in a software program. A software program is a sequence of instructions which defines the operations a CPU has to perform in a specified order.

An FPGA is a parallel device of which the functionality can be configured at device startup. Both software and 'FPGA functionality' are expressed in a human readable language which is compiled to respectively a software program or an FPGA programming file. The main difference is that an FPGA programming file does not contain instructions but it defines a structure of a digital circuit. Depending on performance constraints specified to a HDL compiler more operations are performed in parallel, which takes more FPGA building blocks, or operations are performed sequentially to save FPGA building blocks. If writing software is a 1-dimensional problem in execution time, FPGA programming is a 2-dimensional problem in FPGA area and execution time.

In this report the content which is used to configure an FPGA is called *configware* instead of hardware (Hartenstein, 2006).

Vendors of FPGAs provide pre-built configware blocks in order to save design time. Pre-built configware blocks are called *Intellectual Properties* (IPs) or soft-cores. The last term is used in this report. The term hard-core exists as well and it refers to fixed blocks integrated into FPGAs like DSP blocks or CPU cores.

1.5 Report outline

This report provides some background information on various topics regarding the Xilinx hardware and software. Some of the chapters can be seen as a continuation of these background chapters but they cover the work carried out in this assignment e.g. work in order to get the ML510 board working. The choice was made to place chapters containing own work after the corresponding background chapter.

The second chapter of this report gives an overview of the Xilinx ML510 board. The third chapter explains the work needed to get the ML510 running. The fourth chapter gives background on the Xilinx design flow and their tooling. The fifth chapter proposes how the Xilinx design flow can be fitted into the CE design flow. The sixth chapter applies the proposed design flow for the creation of a robotic test case. The final chapter draws conclusions and gives recommendations for future projects.

The appendices provide information on how to install Linux and Embryo on the ML510 and give technical details about the robotic demonstrator.

2 ML510

In this chapter an overview of the capabilities of the ML510 is given. The first two paragraphs give a look at the ML510 and the Virtex-5 FPGA from a computer architecture point of view. The third section gives an overview of the User I/O ports of the ML510. The fourth section gives an overview of supported operating systems. The last section compares the Virtex-5 FPGA with FPGAs used in previous CE projects.

2.1 Overview

A picture of the Xilinx ML510 development board is shown in Figure 2.1. At first glance the board looks like an ATX-sized PC motherboard with features common on PC motherboards like PCI slots, IDE ports, DIMM banks and more. Closer inspection reveals that there is no general purpose processor like a x86-processor on the board but a Virtex-5 FX130T FPGA.

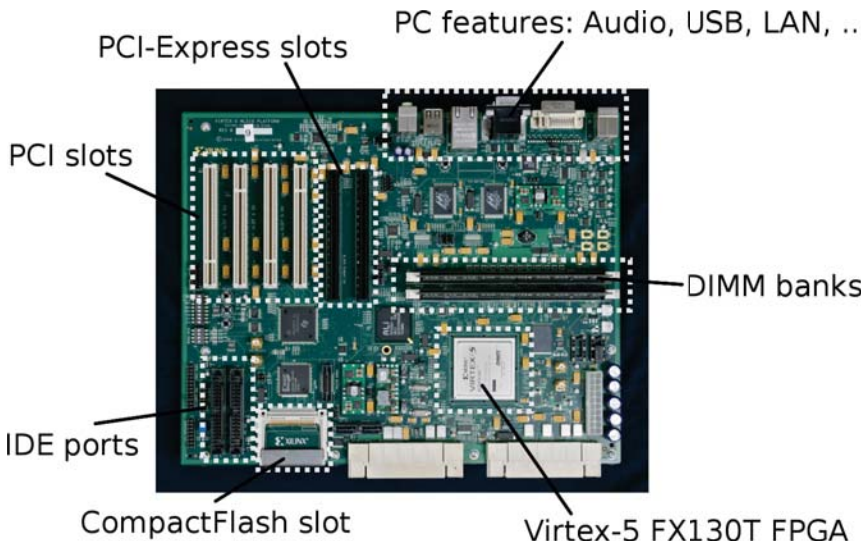

Figure 2.1: Xilinx ML510 FPGA board

General FPGA development boards offer a large number of User I/O pins in combination with communication interfaces like UARTs and Ethernet. The ML510 offers this as well but also offers a large amount of features normally found in PCs. An overview of the features of the ML510 is given below:

- FPGA: Virtex-5 FX130T
- CPU: 2x IBM PowerPC 440 core (inside the Virtex-5 FX130T)
- Memory: 2x DIMM banks for DDR2 memory (by default each contains 512MB
- Expansion slots: 4x PCI, 2x PCI-Express
- Storage: 1x CompactFlash interface, 256Mbit onboard Flash, 2x IDE, 2x Serial ATA
- LAN: 2x Gigabit Ethernet physical layers
- Video: 1x DVI-I, max resolution 640x480
- Audio: AC97 audio with headphone and microphone ports
- I/O-ports: 2x RS232, 2x PS/2, 2x USB 1.1
- User I/O: 124 GPIO pins, 4x debug LED/switch

As can be seen in the list, the ML510 offers a large amount of functionality. Most of this functionality is not realized using dedicated hardware but by soft-cores inside the

FPGA. Xilinx and their partners ship a large number of different soft-cores which can be flashed inside the FPGA. This way the ML510 can be configured to the requirements of the designer.

In case of Ethernet: the ML510 contains two onboard Ethernet physical layers. Different Ethernet soft-cores are available, which allow for realization of different types of Ethernet cards. Two of the available Ethernet soft-cores are a Gigabit Ethernet core by Xilinx and a real-time Ethernet core EtherCAT core by Beckhoff (Beckhoff, 2008).

2.2 Virtex-5 FX130T FPGA

This section gives an overview of the Virtex-5 FX130T FPGA and two of its components namely DSP blocks and the embedded IBM PowerPC 440 processor cores.

2.2.1 Overview

Xilinx offers two families of FPGAs: the low-cost Spartan and the high-end Virtex. A recent Virtex family is the Virtex-5. It is made using 65nm technology and it offers various architectural improvements over previous Xilinx FPGAs.

The most significant change in the Virtex-5 over previous Xilinx FPGAs is the design of its logic cells. In previous Xilinx FPGAs, these were made up of a 4-input lookup-table (LUT) in combination with a flip-flop and some other logic. The Virtex-5 uses larger logic cells, each consisting of 6-input LUT paired with a flip-flop and the other logic. Four logic cells are grouped together in slices.

A 6-input LUT has a capacity of 2^6 entries which is equivalent to four traditional 4-input LUTs, which have 2^4 entries each. Not all designs can take advantage of the larger LUTs. For example basic bitwise operations which need less than six inputs, do not use a 6-input LUT up to its maximum capacity. Xilinx claims that on average one Virtex-5 logic cell is equivalent to 1.6 traditional logic cells (Xilinx, 2007). This factor will be used throughout this report when comparing resource usage with previous FPGA designs.

Within the Virtex-5 family, there are five product lines each optimized for a different application. The differences can be found in the number of logic cells, the number of DSP blocks and the number of high-speed communication links. The FPGA on the ML510 is a Virtex-5 FX130T (see Figure 2.2) which belongs to the FXT product line. This product line is optimized for embedded processing and offers one or more embedded IBM PowerPC 440 processors in combination with a moderate amount of logic cells and DSP blocks. The Virtex-5 FX130T is one of the largest Virtex-5 FXT models.

Figure 2.2: Xilinx Virtex-5 FX130T FPGA

An overview of the features of the Virtex-5 FX130T is given below:

- Slices: 20,480 (contains logic cells and flip-flops)
- Logic cells: 81,920 6-input LUTs (equivalent to 131,072 traditional 4-input LUTs)
- Flip-flops: 81,920
- Maximum operating frequency: 550MHz
- DSP blocks: 320x DSP48E
- Processor block: 2x IBM PowerPC 440 cores
- Block RAM: 298 blocks of 36kbit for a total of 10,278kbit
- Hard-core memory controller with support for DDR2 using soft-core bridge
- LAN: 6x 10/100/1000 Ethernet MAC blocks
- High-speed serial I/O: 20x RocketIO transceivers capable of 150Mbps to 6.5Gbps

2.2.2 DSP48E block

Algorithms make frequent use of arithmetic operations like additions, subtractions and multiplications. These operations can all be implemented in logic cells, but the use of logic cells has two disadvanages. First, when a high number of bits is needed (e.g. for accuracy) a large number of logic cells is required. Second performance of a design implemented solely using logic cells is limited. This is because a structure like a multiplier can require hundreds of logic cells which all have to be connected. Connecting these using an internal network reduces performance. For this reason DSP blocks which can perform arithmetic and logic operations have been added to modern FPGAs.

The Virtex-5 FX130T contains 320 DSP48E blocks which can operate at a frequency upto 550MHz. The DSP blocks are spread in columns and rows over the FPGA. Within a column DSP blocks are connected to each other using an internal network.

A DSP block (see Figure 2.3) consists of a 2-complement 25x18 multiplier in parallel with an integrated 48-bit ALU and logic for buffering and signal routing.

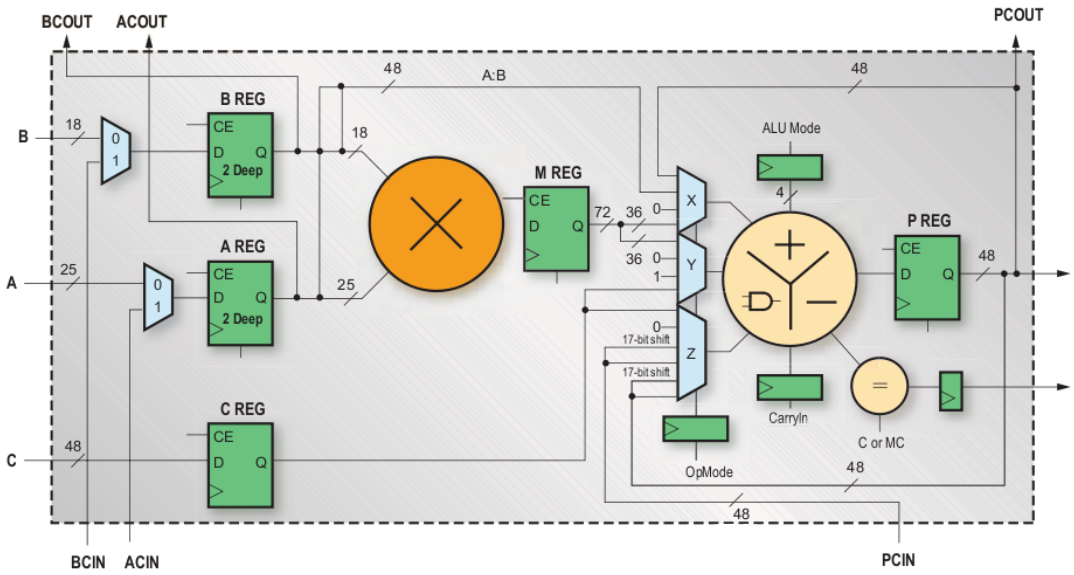

Figure 2.3: Virtex-5 DSP48E block, image (Przybus, 2007)

A DSP block receives input data from either user driven inputs A, B and C or from other DSP blocks using ACIN, BCIN and PCIN. The input source and the operation of the ALU can be selected using multiplexers. The ALU can perform additions, subtractions and various logic operations. It can perform these operations on input data from C and

PCIN or from the output data of the multiplier. It can also be fed with previous results stored in the P register and in this way it acts like an integrator.

There are three ways using which DSP blocks can be used in a design. First, smart FPGA synthesis tools are able to automatically realize logic and arithmetic operations using DSP blocks when it makes sense to them. Second, the designer can explicitly instantiate DSP blocks in his design but this requires additional work. The work includes configuration of multiplexers and other logic in the DSP block. Third, design tools can offer wizards for the creation of filters and floating-point cores which use the DSP blocks internally.

The DSP block can be used in various applications, two of these are explained here. One such an application is the creation of digital filters. Digital filters consist of a large number of stages and each stage performs multiplications and subtractions. Each stage can be mapped on a single DSP block. Using PCIN and PCOUT multiple stages can be chained together.

A second application of the DSP block is the implementation of floating-point. Xilinx provides a floating-point core wrapper named *Floating-Point Operator* (Xilinx, 2009a) which can be used for implementing single and double-precision floating-point. For single-precision, the floating-point wrapper requires two DSP blocks and the wrapper can operate upto 410MHz. Each calculation requires multiple clock cycles. For instance a multiplication takes 8 clock cycles and an addition takes 11. The effective operating frequency of the wrapper is approximately 50MHz (=410MHz/8) for multiplications and 40MHz (=410MHz/11) for additions. The effective operating frequency can be increased by using more DSP blocks. For example, doubling the operating frequency requires twice the number of DSP blocks.

2.2.3 IBM PowerPC 440 CPU core

The Virtex-5 FX130T features two separate IBM PowerPC 440 cores. PowerPC is a RISC based processor architecture developed by IBM in coorperation with Apple and Motorola (IBM, 2009). These days PowerPC based CPUs are used in all sorts of devices ranging from consumer devices, like Wii, Playstation3 and Xbox360 to high performance servers and embedded systems. For integration into SoCs, IBM offers embedded PowerPC cores. This type is used in the Virtex-5 FXT series.

The PowerPC 440 is a 32-bit big-endian CPU, based on the PowerPC Book-E specification for embedded processors (IBM, 2003). This specification defines a common subset of the PowerPC architecture which all Book-E designs must implement. Extra functionality like 64-bit support or a hardware floating-point unit is optional.

The Book-E specification also defines an interface for an *Auxiliary Processor Unit* (APU) which can hook into the execution engine of the processor for the creation of custom CPU instructions. Xilinx for instance offers a PowerPC-compatible soft-core floating-point unit which makes use of it.

A list of features is shown below.

- Architecture: 32-bit PowerPC (RISC)
- Endianness: Big-Endian
- Clock frequency: upto 475MHz [1]
- Integrated memory management unit
- APU-bus for creation of custom CPU instructions
- Floating point: available using APU-connected soft-core

Xilinx has added logic around the PowerPC 440 core for high-performance, low latency

[1] In the FX130T FPGAs with the fastest speed grade PowerPC 440 cores can run upto 550MHz, the ML510 uses a lower speed grade limited to 475MHz

access to system memory and periperhals inside the FPGA. The resulting hard-core embedded processor block is shown in Figure 2.4. The block consists of the PowerPC 440 core, an APU-bus for interfacing with co-processors, *Device Control Registers* (DCR) and a Crossbar interface.

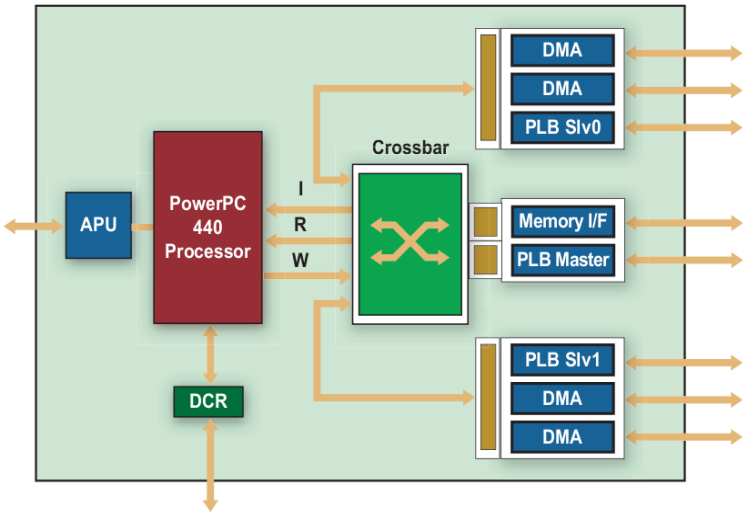

Figure 2.4: Embedded PowerPC 440 processor block with Crossbar (Abramson et al., 2008)

The Device Control Registers are a special type of registers which are visible from inside the CPU core (and also have their own special CPU instructions) but are actually implemented outside the CPU core as a soft-core. The registers are meant as a control interface for busses and custom peripherals implemented in FPGA logic. Some soft-cores like the Xilinx Ethernet core and the Xilinx VGA core use the DCR interface.

The Crossbar is used by both the CPU and the FPGA to access system memory and peripherals. A more detailed diagram of the Crossbar interface which illustrates this is shown in Figure 2.5. The light colored areas are inside the FPGA, the DDR2 memory is outside the FPGA in the ML510 dimm banks. The Crossbar has five input connections (on the left) and two output connections (on the right).

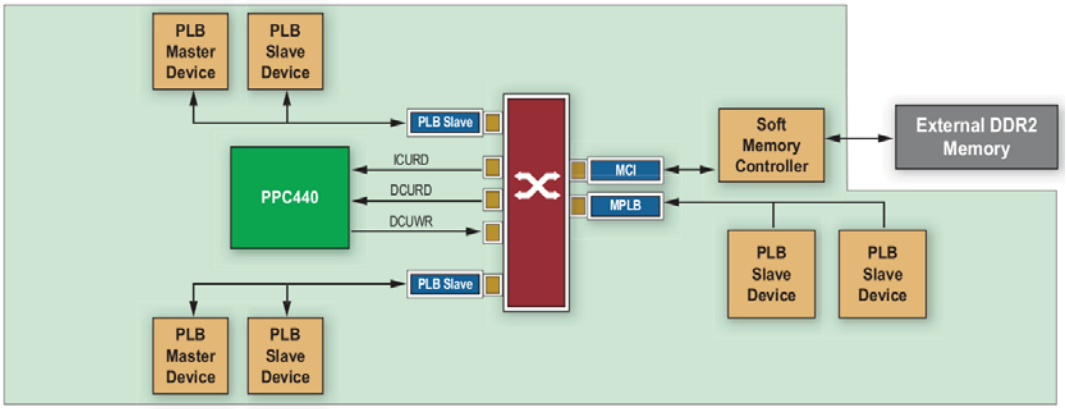

Figure 2.5: Crossbar with peripherals attached (Abramson et al., 2008)

The input connections are so called *Processor Local Busses* (PLBs) and each is 128-bit

wide. Three of the PLB inputs, named ICURD, DCURD and DCURW, are used by the CPU for respectively reading instructions, reading data and writing data. The other two 'PLB Slave' inputs can be used as a PLB as well but they can also be used as a LocalLink DMA port as shown in Figure 2.4. A LocalLink interface is used by high-speed devices like Ethernet cores.

The two output connections are connected to respectively a *Memory Controller Interface* (MCI) and a PLB to which PLB slave devices can be attached.

Peripherals connected to the input connections are called masters. They can initiate data read or write transactions on the Crossbar. As part of a transaction a master provides a memory address. Based on this address, the Crossbar forwards a transaction to either the MCI or a *PLB Master* (MPLB). In case of the ML510, the MCI is wired to DDR2 system memory using a soft-core memory controller. The PLB Master is interfaced to PLB slave devices for GPIO and UART.

Communication between a soft-core in the FPGA and the CPU can be done in various ways. First of all a soft-core can be connected as a PLB slave device to the MPLB. The crossbar is then able to provide a direct connection between soft-core and a master like the CPU. Another method is to use shared memory. For this the soft-core uses a PLB Slave or a LocalLink to access system memory and writes his data there. The CPU can read it there but there would be no direct connection. Performance can be higher because both soft-core and CPU do not have to be synchronized.

2.3 User I/O

The ML510 board has been designed for high-performance embedded applications. For this reason it offers various high-speed communication busses like PCI, PCI-Express, S-ATA and Ethernet. Standard development boards offer pin headers for interfacing with custom peripherals. These pin headers are not suited for high-speed communication at hundreds of megahertz and for this reason Xilinx provides two 'Xilinx Personality Module' (XPM) interfaces named 'PM1' and 'PM2' as shown in Figure 2.6.

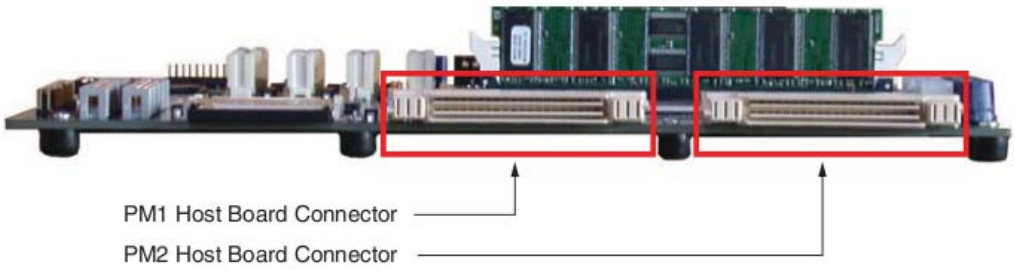

Figure 2.6: ML510 XPM connectors

The connectors as shown in Figure 2.7 are Tyco 'Z-Dok' connectors. They are designed for high-speed differential communication at bitrates upto 6.5Gbps and for this reason they are expensive. In control applications the highest frequent signals are needed for *Pulse Width Modulation* (PWM). When using a PWM frequency of 10kHz, a duty cycle of 1% corresponds to a bitrate of 1MHz, so the Z-Dok connectors can operate at three orders of magnitude higher bitrate than needed.

On the ML510 eight RocketIO transceivers of the Virtex-5 are connected to PM1 and these can operate from 150Mbps to 6.5Gbps. The rest of the I/O pins of PM1 and PM2 are pairs at 2.5V designed for Low Voltage Different Signaling (LVDS). When this signaling method is used, a signal is transported over two wires each carrying the same signal but

with an opposite sign and the signal level is not relative to ground. LVDS is less sensitive to disturbances and for this reason used for high-frequent signals. Each LVDS pair can be used single-ended as well and in this way act like a normal pin. While most pins use 2.5V, a small number of pins is also suited for 3.3V.

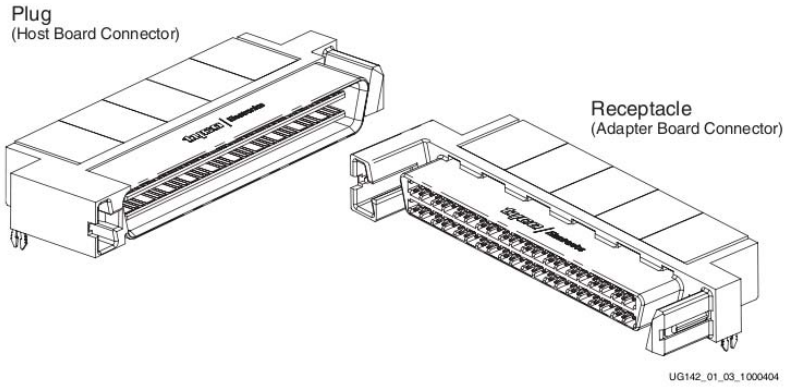

Figure 2.7: Tyco Z-Dok connectors

Both PM1 and PM2 ports combined provide the ML510 with upto 158 I/O pins. The features of each connection are given below. The PM1 connector provides the following signals:

- 8 high-speed RocketIO serial links
- 3 LVDS pairs at 2.5V (can be used as 6 single-ended I/O at 2.5V)
- 1 LVDS clock pair at 2.5V
- 12 single- ended I/O at 2.5V
- 26 single-ended I/O at 3.3V
- 1 single-ended clock at 2.5V
- 1 pin not connected

The PM2 connector on provides the following signals:

- 39 LVDS pairs at 2.5V (can be used as 78 single-ended I/O at 2.5V)
- 1 single-ended clock at 2.5V
- 1 pin not connected

2.4 Operating system support

The availability of a PowerPC 440 CPU inside the Virtex-5 FX130T allows the use of operating systems with PowerPC support. The Xilinx ML510 is officially supported by three operating systems: Xilinx standalone, Xilinx Xilkernel and VxWorks. Other Xilinx boards based on a similar FPGA as used on the ML510 are supported on Linux and QNX as well. Due to support for other Virtex-5 FXT FPGAs in Linux and QNX, there is also a limited amount of unofficial support for the ML510.

As part of this assignment Linux support has been added for the ML510 and this is available in the Linux 2.6.31 kernel (see chapter 3). The performed work includes adding support of peripherals not available on other Xilinx boards which are supported on Linux.

An overview of the features of the different operating systems kernels is shown in Table 2.1. In case of Linux the table reflects the features as supported in Linux 2.6.31. For QNX the table reflects the features supported on the ML507.

The operating systems shipped with the Xilinx development tools are 'Xilinx Standalone' and 'Xilinx Xilkernel'. The first is a minimalistic kernel which is linked into an executable.

Feature	Operating System kernel				
	Standalone	Xilkernel	VxWorks	Linux	QNX
Console	+	+	+	+	+
Memory management	–	–	+	+	+
Timers	–	+	+	+	+
Real-time support	–	–	+	? [2]	+
Multithreading	–	+	+	+	+
Networking	+/–	+/–	+	+	+
Filesystem support	+/–	+/–	+	+	+

Table 2.1: Features of operating system kernels

It can boot the CPU and bring up a console with input and output on the UART. From software there is access to all board periperhals including flash memory, Ethernet and user I/O using user mode libraries for which Xilinx provides a few examples. The Xilinx Xilkernel is similar to Xilinx Standalone but adds support for Posix threads and timers. Both Xilinx Standalone and Xilinx Xilkernel are suited for running basic control algorithms, but since each kernel provides only limited functionality the user needs to guarantee real-time behavior and provide memory management.

VxWorks (Wind River Systems, 2009) is a commercial, closed-source, real-time operating system developed by Wind River Systems, a subsidiary of Intel. This operating system supports a large number of hardware platforms and it is widely used by the industry in aerospace, automotive, military and telecommunication applications.

Linux (Torvalds, 2009) is an open-source, UNIX-compatible operating system with support for a large number of hardware platforms. This operating system is frequently used for real-time applications within CE. This requires the use of real-time patches like RTAI (DIAPM, 2009) or Xenomai (Xenomai Project, 2009) patches. These are also available for the PowerPC 440 architecture but whether they work on the ML510 is not known. Most likely changes in the Xilinx interrupt controller driver are needed.

QNX (QNX Software Systems, 2009) is a commercial, real-time, operating system for use in embedded systems. This operating system uses a micro-kernel which is responsible for task creation, scheduling, interprocess communication and interrupts. All other tasks are handled using 'servers' in user-space, which are applications running in the background. This allows disabling of features without recompiling the operating system. The source code of the QNX kernel is available for non-commercial use.

An overview of the peripherals supported by each operating system is given in Table 2.2. As is shown in the table, Standalone and Xilkernel offer basic I/O support and access to flash storage. VxWorks and QNX add support for LAN. Linux supports most periperhals including including Audio, IDE, PCI, USB and Video.

Peripherals unsupported by any of the operating systems are: PCI-Express, S-ATA and PS/2. For PCI-Express there is a soft-core but no driver for it has been written. In case of S-ATA there is no S-ATA controller core available. The reason the board contains S-ATA connectors at all is because they allow for creation of a high-speed communication link using commercial-of-the-shelf cables. PS/2 is part of a ALI M1533 PCI chipset and depends on PCI support. Linux support for it can easily be added but it is not available at this point.

[2] Real-time Linux patches (RTAI and Xenomai) are available for PowerPC 440 Linux, but whether they work on the ML510 is unknown.

Peripheral		Operating System				
		Standalone	Xilkernel	VxWorks	Linux	QNX
Expansion Slots	PCI	-	-	-	+	-
	PCI-Express	-	-	-	-	-
Storage	CompactFlash	+	+	+	+	+
	IDE	-	-	-	+	-
	Flash	+	+	+	+	+
	S-ATA	-	-	-	-	-
Other	LAN	-	-	+	+	+
	Video	-	-	-	+	-
	Audio	-	-	-	+	-
I/O Ports	PS/2	-	-	-	-	-
	RS232	+	+	+	+	+
	USB	-	-	-	+	-
User I/O	Debug LEDs/switches	+	+	+	+	+
	GPIO	+	+	+	+	+

Table 2.2: ML510 peripheral support in operating systems

2.5 Virtex-5 compared to FPGAs used in previous CE projects

The preceeding sections described the Virtex-5 from a computer architecture point of view. It is described using quantities like logic cells and DSP blocks. This section illustrates the use of the Virtex-5 for control. This is done by estimating the FPGA resource usage of a control design, previously made for a different FPGA, if would be transferred to a Virtex-5. The results of a recent FPGA project on floating-point based control of the Production Cell setup (Sassen, 2009; Groothuis and Broenink, 2009) are used as a reference.

The FPGA used for the control of the Production Cell is a Xilinx Spartan-3 XC3S1500. The Spartan-3 is based on a less modern architecture than the Virtex-5 and it offers less features but at a lower price. A comparison between the two FPGAs is shown in Table 2.3.

Feature	FPGA	
	Spartan-3 XC3S1500	Virtex-5 FX130T
Logic cells	29,952	131,072 [3]
Flip-flops	29,952	81,920
Maximum operating frequency	200 MHz	550 MHz
Processor cores	0	2
DSP blocks	0	320
Multipliers	32	0 [4]
Block RAM	576kbit	10,278 kbit

Table 2.3: Comparison between Virtex-5 FX130T and Spartan-3 XC3S1500

The Production Cell setup consists of six motors and each motor requires a PID controller and a motion profile. The setup uses a sample frequency of 1 kHz. The Spartan-3 did not offer enough FPGA resources for a parallel floating point implementation of six PID controllers. In order to fit the design into the Spartan-3, a single PID controller block was designed in such a way that it could be reused for all six PID calculations. The same was done for the calculation of motion profiles. The results are shown in Table 2.4 (Groothuis and Broenink, 2009).

[3] Equivalent number of 4-input LUTs

[4] On the Virtex-5, DSP blocks contain multipliers

	FPGA	
Element	LUTs (amount)	Flip-flops (amount)
Floating-point library + wrapper	27.4% (8191)	19.7% (5909)
PID controllers	4.2% (1251)	0.3% (91)
Motion profiles	1.1% (314)	0.5% (163)
I/O + PCI	4.1% (1250)	1.8% (534)
S&C Framework	5.6% (1666)	4.2% (1250)
Free	57.6% (17280)	73.5% (22005)

Table 2.4: Estimated FPGA resource usage of the Production Cell on a Xilinx Spartan-3 FPGA

The results in the table are estimated based on compile logs of the Handel-C compiler. As can be seen in the table a large number of LUTs and flip-flops is taken by a floating-point library. The design contains this library for carrying out the majority of the calculations for both the PID controllers and motion profiles. The compiler has placed a large portion of the FPGA resources which were only used in the PID controller or Motion profile code to the floating point library. For estimating how many times the design can fit into the Spartan-3 the FPGA resource usage of these three parts needs to be added together. In total the sequential design can fit the Spartan-3 twice but it is not big enough to fit it six times.

The Production Cell design could be moved changes in the pin-out to the Virtex-5. This recompilation has been attempted but during synthesis a lot of compile errors were encountered which would take a lot of effort to fix. Instead, the results are calculated out of the Spartan-3 results using the approximation that one 6-input Virtex-5 LUT equals 1.6 Spartan-3 LUTs. This means that the amount Spartan-3 LUTs has been divided by 1.6, the number of flip-flops stays the same. The results are shown in Table 2.5.

	FPGA	
Element	LUTs (amount)	Flip-flops (amount)
Floating point library + wrapper	6.3% (5120)	7.2% (5909)
PID controllers	1.0% (780)	0.1% (91)
Motion profiles	0.2% (200)	0.2% (163)
I/O + PCI	1.0% (780)	0.7% (534)
S&C Framework	1.3% (1040)	1.5% (1250)
Free	90.2% (74000)	90.3% (73973)

Table 2.5: Calculated FPGA resource for the Production Cell using a Xilinx Virtex-5 FPGA

The table illustrates that for a naive translation, only 10% of the capacity of the Virtex-5 is needed. Using DSP blocks even a much lower usage can be obtained because the floating-point library can be reduced to a DSP wrapper library consuming mostly DSP blocks and a small amount of LUTs and flip-flops. As explained in subsection 2.2.2, two DSP blocks can be used for implementing single-precision floating-point. Below the number of PID calculations which can be performed in one sample period of 1ms is calculated. This allows for comparing DSP blocks with the Production Cell design. It is considered that on average a single floating-point calculation takes 10 clock cycles and that the DSP blocks can operate at 410MHz, see subsection 2.2.2 . This results in an effective operating frequency of 41MHz. The discrete 20-sim PID controller block requires approximately 20 computations. This means that in theory two DSP blocks can carry out approximately

2,000,000 PID calculations a second or 2,000 each sample period (1ms). The Virtex-5 contains 320 DSP blocks, so it allows for 320,000 PID calculations per sample period or 320,000,000 per second!

The motion profiles could also be realized using DSP blocks and this would save additonal LUTs and flip-flops. An even better way would be to store motion profiles in memory blocks which are filled by the PowerPC core at startup. This way only a small number of LUTs and flip-flops would be needed to create a bus interface to the processor, the rest would be located in block memory of which the Virtex-5 offers a large amount. An advantage of this approach would be that for a motion profile with a different shape only the software needs to be changed and no changes to the FPGA design have to be made.

The described redesign of the Production Cell using Virtex-5 functionality would require only a few percent of LUTs and flip-flops since PID controllers and motion profiles can be implemented efficiently using DSP blocks and block memory. In total 320,000 PID calculations could be calculated in a single sample period which is enough for 320,000/6=53,000 Production Cell setups. Though at some point the number of User I/O pins (158) becomes a limitation and serial communication busses e.g. EtherCAT must be used.

By these calculations, it is demonstrated that the Virtex-5 is overkill for solving basic control problems. Computation intensive calculations take better advantage of the Virtex-5. Vision for robotics is one such application as was mentioned in the introduction. The Virtex-5 would be able to carry out all needed computations within a control loop.

3 Preparation of the ML510 for control applications

The first half of this assignment is devoted to making the ML510 usable for CE projects. This work consisted of programming, hardware testing and documenting all steps on the CE wiki.

The first section describes the work carried out for adding ML510 support to the Linux kernel. The second section describes integration of the ML510 Linux code with Embryo. The third section describes work on an external I/O board which provides the ML510 with I/O pins for interfacing with external periperhals like a plant.

3.1 Linux kernel

Linux is the most used operating system at CE for use in embedded control systems. The availability of Linux support for the ML510, as claimed at the launch of the ML510 (Xilinx, 2008b), was one of the reasons for acquiring the board. After the board had arrived and tests were performed, it appeared that no Linux support was publicly available for the ML510.

Xilinx had written ML510 Linux support for an outdated Linux kernel but they decided not to release the code because it was incomplete and would have to be rewritten for use with modern Linux kernel versions which they decided not to do so.

Besides the ML510, Xilinx offers other FPGA boards based on the Virtex-5 FXT. One of these is the ML507 which contains a smaller FPGA with a single PowerPC 440 core and less onboard features compared to the ML510 for instance it lacks PC functionality like IDE, PCI, and VGA support but it does provide standard pin headers for I/O. Xilinx provides official support for the ML507 in Linux.

Due to similarities between the ML507 and the ML510, it was decided to port Linux to the ML510 and take the ML507 code as a starting point. In order to prepare the ML510 for vision, which was the intended test case for this assignment, work was also done on adding support for the PC features offered by the ML510.

The performed ML510 Linux work includes:

- Creation of a Linux-compatible configware package for the ML510.
- Adopting ML507 Linux code for use on the ML510.
- Writing of a PCI driver for the ML510 to make previously unsupported board functionality available.
- Submission of the ML510 Linux work to the Linux developers for inclusion in the official Linux kernel distribution.

The remaining of this section describes the ML510 Linux work in detail. Instructions for installing Linux on the ML510 can be found in Appendix B.

Before the ML510 is able to run software, configware needs to be loaded into it for interfacing the CPU to a memory controller and setting up board peripherals. For the ML507, Xilinx provides a pre-built configware package to boot Linux. For the ML510 such a Linux-compatible configware package had to be developed from scratch, see Appendix B.1.

The process of porting the ML507 Linux code to ML510 took several weeks. The main difficulty in getting Linux to work is that both the configware and the software portion need to function correctly before Linux starts. A large number of issues had to be addressed before the Linux kernel completely booted. The issues can be classified into two categories:

- No debug output shown on the console
  The most common issue is that no output appears on the UART which is used as a serial console for input and output. In general this issue is caused either by incorrect system information provided by a DTS-file to the Linux kernel or by a Linux kernel panic which occurs before the console is initialized. The first issue can be fixed by verifying by hand whether the DTS-file is correct. The second issue is hard to debug because no debugger is available, which means that the designer needs to act as a debugger. In this project special UART debug code was added to the Linux kernel boot process to reveal the kernel panic log.
- Linux kernel panic
  A Linux kernel panic occurs when a driver or other part of the Linux kernel crashes. In general such issues are caused by driver bugs caused by null pointer exceptions or deeper issues. The kernel crash log and related driver code needs to be analyzed to find the root of the problem.

The two categories of issues described above give an idea about the issues encountered. In general debugging crashes on the ML510 is very time consuming and depending on the issue it takes hours or even days.

In order to prepare the ML510 for robotic applications a PCI driver was written to make previously unsupported PC functionality of the ML510 available. The driver allows the use of the on-board PC functionality but it also allows the use of PCI add-on cards e.g. a FireWire add-on card for interfacing with a video camera. Details on the PCI driver can be found in Appendix A.2.

The finished ML510 Linux code has been submitted for inclusion into the official Linux kernel distribution and is part of the Linux 2.6.31 which is to be released in September 2009. The inclusion of the code in the Linux kernel is important because it ensures that the code gets maintained and it proves the quality of the written code. Before any code enters the Linux kernel, it is reviewed by other developers and they provide feedback for improving the code. In general, a number of design iterations is needed before the code is considered clean enough for inclusion in the Linux kernel. In this case three iterations were required.

In total it has taken approximately three months for writing all ML510 Linux code. The process of getting the code integrated into the official Linux kernel distribution took another two months.

3.2 Embryo

Within CE a variety of different hardware platforms is used. Most of these run a Linux distribution named Embryo developed by ir. M. Groothuis from CE. Embryo is a small, embedded Linux distribution build on top of OpenWrt (OpenWrt, 2009) which is frequently used on wireless routers. As part of this assignment ML510 support has been added to Embryo. Below Embryo and the ML510 Embryo work are described. Installation instructions for Embryo can be found in Appendix B.

Embryo itself is a set of scripts and configuration files which automate all compilation tasks for building an embedded Linux distribution. These tasks include:

- Downloading source code of compilers, the Linux kernel, libraries and tools.
- Building of cross-compilers.
- Configuration and building of the Linux kernel.
- Compilation of Linux libraries and tools.
- Creation of a Linux root file system.

The Embryo scripts must be used on a development PC running a UNIX-like operating

system like Linux or Mac OS X. The first task is to configure the Embryo distribution. This includes the selection of hardware platform, a set of libraries and the tools to be bundled with the Embryo installation. After that Embryo downloads and compiles the source code of compilers, the Linux kernel and the selected libraries and tools. The compiled Linux kernel, libraries and tools can then be installed on a memory card or a hard disk for loading on the embedded hardware platform.

In case of the ML510, Embryo already had support for the PowerPC 440 architecture. Most work focused on configuration files for a ML510 Linux kernel and integration of a compiled Linux kernel with an FPGA programming file to allow the ML510 to launch Embryo. Details on this integration can be found in section 5.3.

3.3 ML510 I/O board

As explained in section 2.3, the ML510 lacks I/O pin headers which are needed for interfacing with custom peripherals like a plant. An extension board is needed to interface the Z-Dok connectors of the ML510 to I/O pins. Such a board has been designed by the Electronic Systems group of the Technical University of Eindhoven (TU/e) because they have also required access to I/O pins on a ML510 board. An empty PCB of the ML510 I/O board has been acquired from the TU/e. This section describes the board and the work needed to get it working for a CE setup.

The demonstrator which had to be interfaced to the ML510 at the TU/e, was a setup similar to the Production Cell. Originally this setup was controlled by a PLC and now it needs to be controlled by an FPGA. Their requirements for the I/O board were:

- Digital high level of 24V
- Signal frequency in the order of 1 kHz
- Prevent the test setup from damaging the FPGA

Based on these requirements an I/O board was built which provides access to 78 pins of which one half is for input and the other half for output. The board uses optocouplers to isolate the FPGA from the setup. The type of optocouplers used in the design is meant for transmission of analog signals with an analog bandwidth of 100kHz. For use in setups as used within CE, the optocouplers are not ideal because PWM signals which are used for digital-to-analog conversion require a frequency of several MHz for high accuracy. Other types of optocouplers, optimized for digital I/O at high frequencies, are available but they use a different pinout and require different biasing, so they cannot be used on this board.

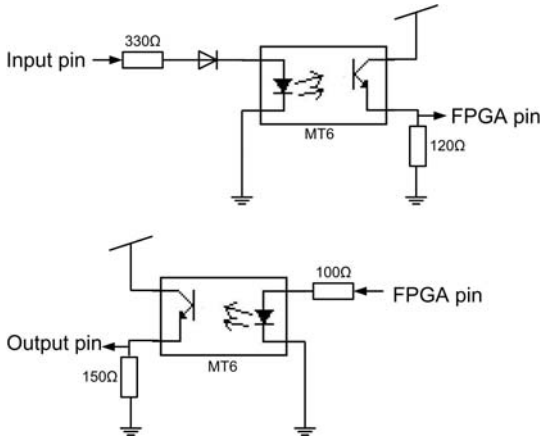

Figure 3.1: Diagram of the circuitry on the ML510 I/O board

For interfacing with a CE setup modifications were made to the circuitry to get it to operate at 5V and component values were adjusted to improve performance of the optocouplers for use with PWM signals. A diagram of the circuitry is shown in Figure 3.1. With these modifications the I/O board is usable but the optocouplers severly limit the accuracy of input and output signals. For instance when outputting a square wave at a frequency of 100kHz with a 50% duty cycle, the signal is high for 55% instead of 50%. This results in losses in accuracy during digital-to-analog-conversions. Similar issues occur for the input pins. Another problem, only limited to inputs pins of the I/O board, is that optocouplers have current inputs while within CE voltage signals are used. A buffer chip is required to be able to drive the optocouplers.

To summarize the ML510 I/O board works but it does not perform well at the signal frequencies required for CE but it is better than not having access to any I/O pins. A different board should be designed within CE which fixes the performance limitations. Until that time, the board can be used as it is or the optocouplers can be replaced with transistors. The FPGA then is not galvanic separated from a test setup but the FPGA pins are still protected and I/O performance for high-frequent signals should be good.

4 Xilinx Design flow

The first section of this chapter introduces the concept of hardware/software co-design.

Xilinx offers various FPGA design tools and multiple design flows (using their tools) for hardware/software co-design. For the creation of designs consisting of an embedded CPU and configware, the design flow recommended by Xilinx is build around a tool named *Platform Studio Embedded Development Kit* (EDK). Development of software (SW) and configware (CW) takes place in specialized tools outside EDK, the EDK is used for their integration into the overall system design. The EDK is described in section two.

The third section describes *Platform Studio SDK* which is used for software development.

Xilinx offers two tools for the development of configware namely *ISE Foundation* and *System Generator for DSP* (SysGen). The first tool, described in section four, is the one recommended by Xilinx for obtaining the best performance and lowest FPGA resource usage at the cost of a large design effort. The second tool, described in section five, requires a smaller design effort at the cost of performance and FPGA resource usage.

4.1 Hardware/Software co-design

When creating a design which consists of a CPU and an FPGA, decisions have to be made on which parts of an algorithm to implement in configware and which in software. Good candidates for implementation in configware are the parts which are either computation intensive or which have hard real-time requirements.

The process of paritioning an algorithm in configware and software is illustrated in Figure 4.1. It starts with a specification of the algorithm for example in C, MATLAB, Simulink or 20-sim. Next, the algorithm should be analyzed by the designer to discover timing critical or CPU-intensive parts. A profiling tool can obtain statistics on how much CPU time is spent in different parts of the algorithm. This is done by analyzing the algorithm when it is running on a CPU. In the end, the decision on how to partition the algorithm lies by the designer.

When a partioning in configware and software has been made, both parts can be implemented using respectively a hardware description language (HDL) and a programming language. During development co-simulation can be employed to test the individual parts against the algorithm specification or against each other. Co-simulation (Damstra, 2008; Colenbrander et al., 2008) is a type of simulation in which models can be simulated together while each uses a different simulation engine. This can prevent late integration problems.

Xilinx provides forms of co-simulation in their tools. They define a simulation in which parts of a design are executed on an FPGA and parts in a simulator 'hardware co-simulation'. Within the Control Engineering group this is considered a form of 'X'-in-the-loop-simulation (Visser et al., 2004; Michalek et al., 2005; Oosterom, 2006). Oosterom has distinguished four types for mechatronic design as shown in Figure 4.2. In the Hardware-in-the-loop case electrical signals like PWM are used for I/O instead of simulated signals and for that reason a sine wave is shown in the figure.

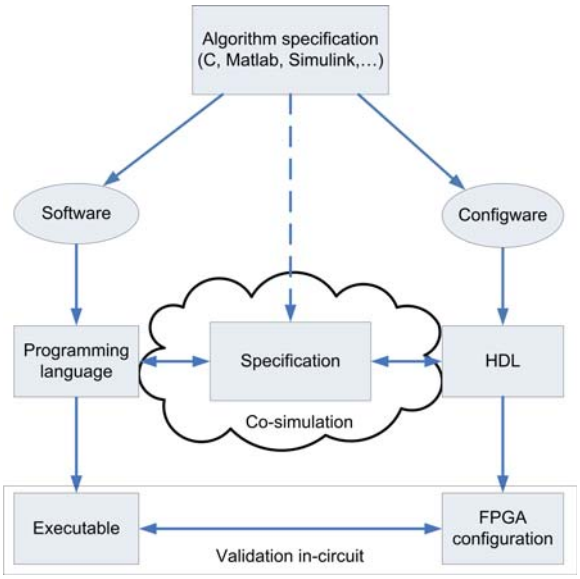

Figure 4.1: Partitioning of an algorithm in configware & software (Colenbrander et al., 2008)

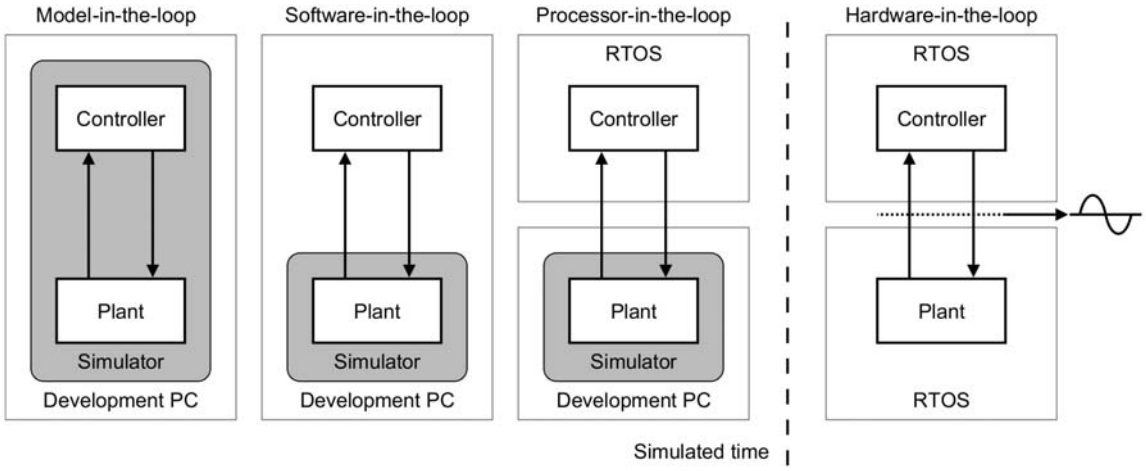

Figure 4.2: Different 'X'-in-the-loop simulation types, image from (Oosterom, 2006)

The four types of 'X'-in-the-loop are described below:

- Model-in-the-loop: models of a plant and control algorithm are connected in a loop and are simulated on the same development PC, this is normal simulation.
- Software-in-the-loop: control algorithm runs as an executable on the development PC, plant and I/O are simulated on the development PC.
- Processor-in-the-loop: control algorithm runs as an executable on the target processor, plant and I/O are simulated on the development PC.
- Hardware-in-the-loop: control algorithm runs on target processor, non-simulated I/O over wires, plant simulated on the development PC in real-time.

Hardware co-simulation in combination in which e.g. the controller is executed on the ML510 and the model on a development PC could be called 'FPGA-in-the-loop' in CE-terminology as illustrated in Figure 4.3.

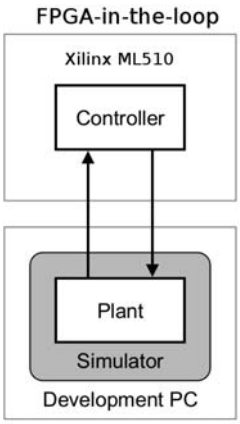

Figure 4.3: FPGA-in-the-loop simulation

4.2 Hardware/software co-design using Platform Studio EDK

The primary Xilinx design tool for hardware/software co-design is Platform Studio Embedded Development Kit. The program is used for overall system development and project management. Development of software and configware takes place in specialized tools, the EDK is used for their integration into the overall system design.

The program ships with a collection of templates and configware for the creation of an embedded design. A new foundation for an embedded design is created using a wizard called *Base System Builder* (BSB). At the start of the BSB wizard a template of the target FPGA board, for example the ML510, has to be loaded. This template provides the wizard with information about the target board (e.g. available CPU, available configware and pin locations). The BSB wizard can then be used to configure the target board with a CPU and the peripherals needed to obtain a bootable system. All hardware information is stored in a *Microprocessor Hardware Specification* file (MHS).

The base system can be extended with user created configware or with configware shipped with the EDK. Additional configware needs to be integrated into the overall system design. Integration consists of connecting communication busses, the assignment of memory addresses and mapping of input/output pins to FPGA pins. The EDK automates most of this integration work but pin mapping has to be done manually. This is done by adding location constraints, which are the mappings of FPGA nets to FPGA pins, to a *User Constraints File* (UCF).

The EDK can be used to set up a simulation environment in which the completed system can be simulated. Different types of simulation can be carried out ranging from fast but inaccurate behavioral simulations to detailed timing-accurate simulations. The simulation time increases with the amount of detail requested. In order to decrease simulation time, the EDK ships with simulation models of busses, configware and processors but even then simulation time can be long. It is recommended to only verify the behavior of individual, custom designed cores and to verify behavior of the complete system on-target.

The ML510 contains a hardware based boot loader named *System Advanced Configuration Environment* (SysACE). Its task is to initialize the FPGA and CPU at board startup. SysACE boots the ML510 from ACE-files which are stored on a CompactFlash card. An ACE-file is the integration of pre-compiled software in the *Executable and Linkable Format* (ELF) with a *FPGA bitstream programming* file (BIT).

The design flow for using the EDK in conjunction with external tools for the development of software and configware is shown in Figure 4.4 (Xilinx, 2008a). The figure shows

three columns each representing a different program. The steps performed using EDK are numbered 1.x and the software and configware blocks have respectively the numbers 2 and 3. These last two numbers are also used for the numbering of steps carried out in software and configware design flows presented in the rest of this chapter and in chapter 5. The numbering schemes used for respectively software and configware are [2.y.x] and [3.y.x], where 'y' is used for numbering the design flow and 'x' for numbering a step within the specific design flow.

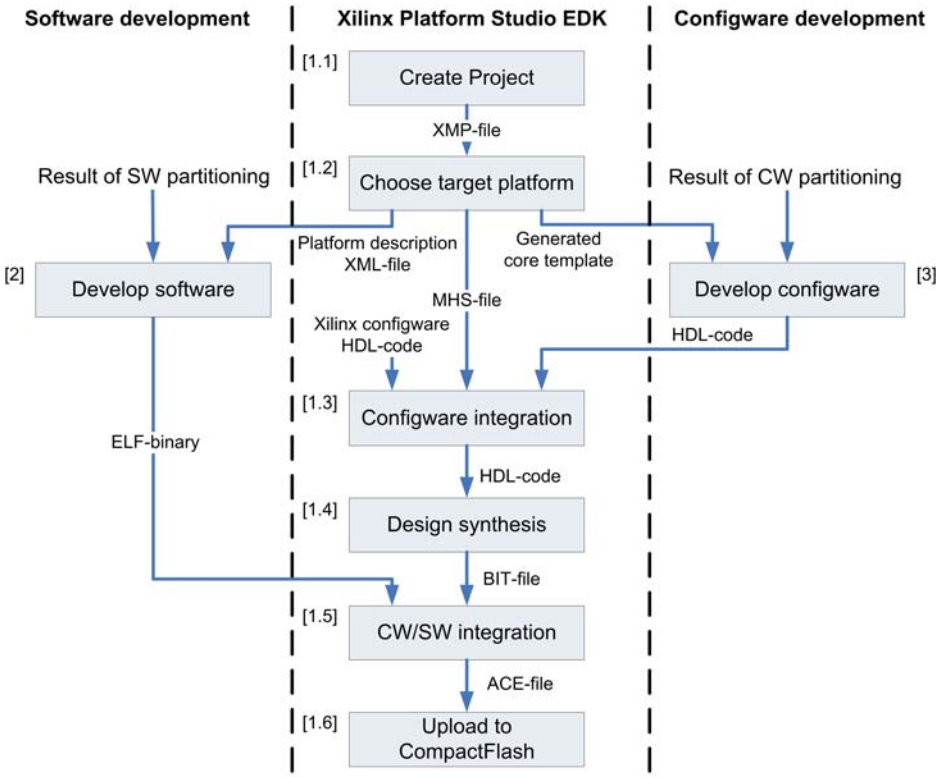

Figure 4.4: Recommended Xilinx EDK design flow

The steps in the design flow are described below:

- Create Project [1.1]
  The design flow begins with the creation of a project in EDK. This results in a *Xilinx Microprocessor Project* file (XMP) which is used for project management.
- Choose target platform [1.2]
  The BSB wizard is used to select an FPGA development board and to configure it with a CPU and the minimum amount of Xilinx provided configware needed to obtain a bootable system.
- Develop software [2]
  Software is developed using a software development tool.
- Develop configware [3]
  Configware is developed using a configware development tool.
- Configware integration [1.3]
  During this phase, wrapper HDL-code is generated using the MHS-file. This wrapper code is the 'main-function' which instantiates all selected configware.
- Design synthesis [1.4]
  During this stage the HDL-code is translated to a netlist and mapped on FPGA building blocks.

- Configware/Software integration [1.5]
  The contents of the ELF-binary is combined with the BIT-file in order to generate an ACE-file.
- Upload to CompactFlash [1.6]
  The final step consists of uploading the ACE-file to a CompactFlash.

4.3 Software development using Platform Studio SDK

Platform Studio SDK is a software development environment for Xilinx devices based on Eclipse (Eclipse Foundation, 2009). Eclipse is a software development environment consisting of an IDE and a plug-in framework for extending it. Originally, Eclipse was designed for software development in Java, using plug-ins it has been extended to support other programming languages (e.g. C, C++ and Python) and other tasks like LaTeX editing or modelling in UML.

Xilinx has used the plug-in mechanism offered by Eclipse to extend it with functionality for MicroBlaze and PowerPC development. This includes the creation of *Board Support Packages* (BSPs), uploading of software to a processor using JTAG and on-target debugging/profiling using JTAG.

Most functionality offered by Platform Studio SDK including C/C++ compilation, JTAG uploading/debugging and more is limited to Xilinx Standalone and Xilkernel operating systems. When a different operating system like Linux is used a different design tool is needed for instance the standard version of Eclipse.

The design flow, recommended by Xilinx, for using Platform Studio SDK in conjunction with the EDK is shown in Figure 4.5. The figure can be seen as a realization of block [2] of Figure 4.4 which contains the EDK design flow. In this section numbering in the form 2.a.x is used for substeps of [2], the letter 'a' is used because chapter 5 describes another software flow. For reference, three steps of the EDK flow have been drawn with a white background color.

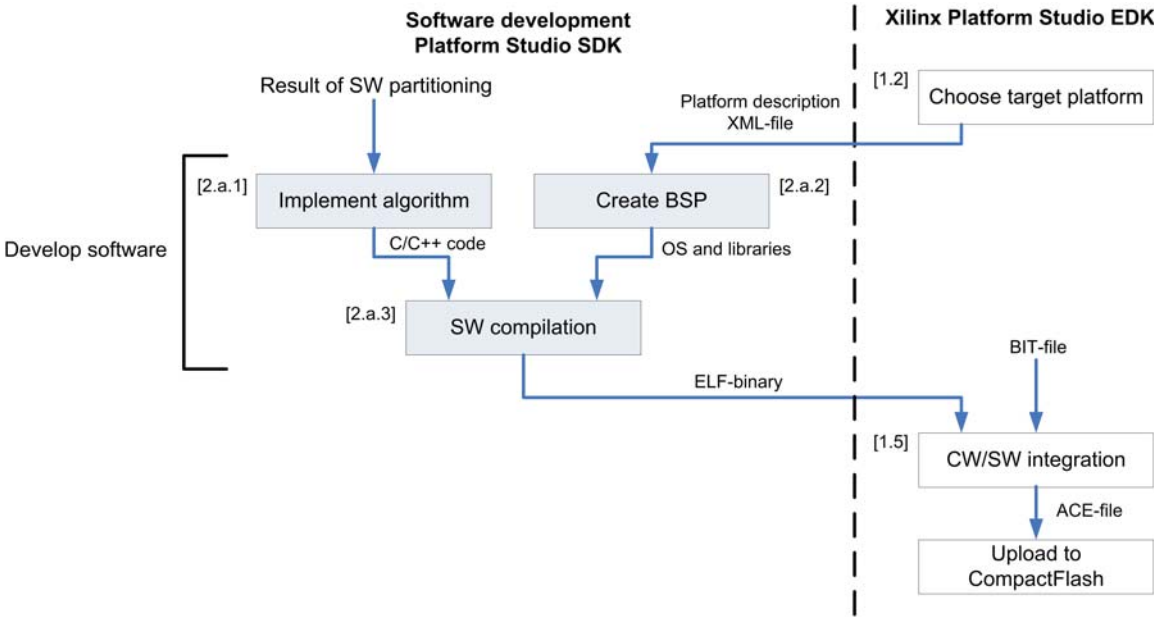

Figure 4.5: Platform Studio SDK software development flow

The steps of the design flow are explained below:

- Implement algorithm [2.a.1]
  The software parts of the algorithm are implemented in C or C++.
- Create BSP [2.a.2]
  The EDK provides the SDK with an XML-file containing relevant hardware information. A BSP-generator extracts relevant information from the XML-file and bundles this with the source code of an operating system and drivers to create a BSP. Optionally the BSP-generator can also add system libraries.
- SW compilation [2.a.3]
  The code of the algorithm is compiled using a cross-compiler and linked with headers and libraries from the BSP. This results in a binary in the ELF-format for execution on a PowerPC or MicroBlaze processor.

4.4 Configware development using ISE Foundation

Xilinx ISE Foundation is a design tool for the development of configware. It is used for writing HDL-code, simulation and the synthesis of HDL-code to a BIT-file.

Development of configware for an FPGA is different compared to software design. First of all development takes place using a HDL instead of a programming language like C/C++. The most important difference is that a HDL is inherently parallel while C/C++ code is sequential. This requires a different way of thinking and writing code. Second, development using a HDL requires knowledge of computer engineering including digital logic design. Third, configware development consists of longer design iterations due to additional, time-consuming development steps.

ISE Foundation offers support for the synthesizable subsets of the HDLs: VHDL and Verilog. Both of these languages can be used to create high-performance designs which make efficient use of FPGA resources. Though development using VHDL or Verilog, which are low-level compared to Handel-C, can be time consuming.

The recommended Xilinx configware development flow using ISE is shown in Figure 4.6. The figure is a realization of block [3] of Figure 4.4. In this section numbering in the form 3.a.x is used for substeps of [3]. The letter 'a' is used because more configware design flows are described in this chapter and in 5. The names of blocks 3.a.1 to 3.a.3 correspond to the names used by Xilinx in the documentation of ISE Foundation. All verification steps are carried out using ISE Foundation or QuestaSim.

The steps in the design flow are described below:

- Design Entry [3.a.1]
  The algorithm is implemented using a HDL in ISE. The functional behavior of the HDL-code is verified by simulation.
- Design Synthesis [3.a.2]
  The functional HDL-code is captured to structural HDL-code which contains a representation of the design at register transfer level (RTL). The structural HDL-code needs to be simulated because its behavior can differ from the function HDL-code.
- Design Implementation [3.a.3]
  The netlist is mapped on FPGA building blocks. If design implementation succeeds the end-result is a BIT-file. In all cases a report is generated which contains timing information and FPGA resource usage. The timing information can be used to refine the design if parts do not meet their timing constraints. Further the timing information can be used during simulation for correct set-up and hold times of logic. This way more potential design errors can be found at an early stage.
- Upload to board [3.a.4]
  The complete design or parts of it can be uploaded to the target board for testing.

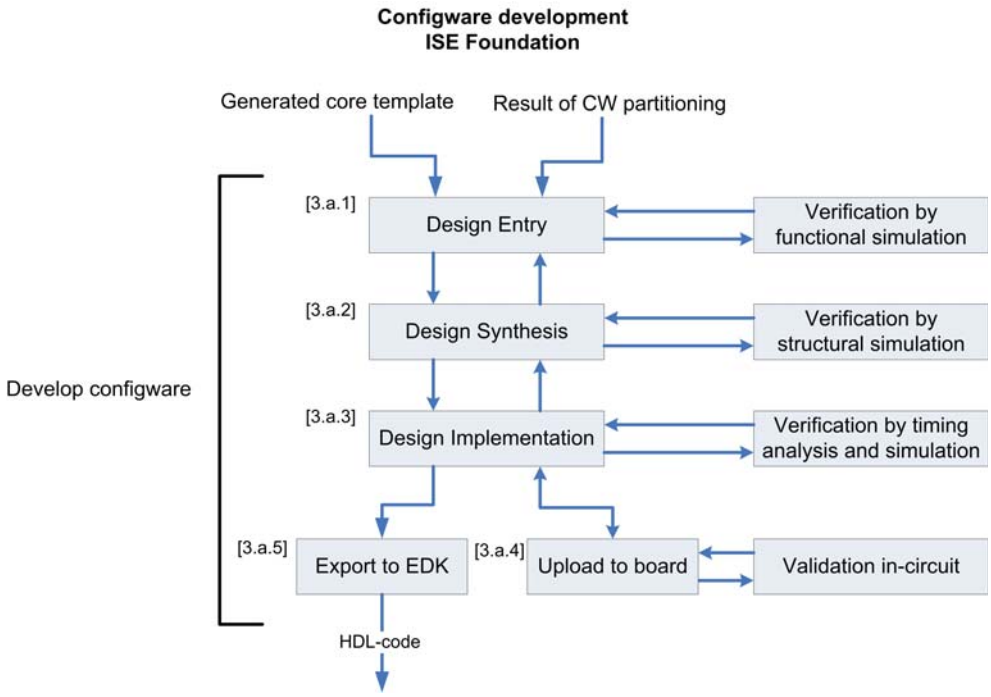

Figure 4.6: ISE Foundation configware development flow

- Export to EDK [3.a.5]
  The HDL-code of the finished configware can be exported back to the EDK for integration with CPU and software.

4.5 Configware development using System Generator for DSP

Besides ISE Foundation, Xilinx offers a second design tool for configware development named *System Generator for DSP* (SysGen). It is a model-driven design tool and compared to ISE Foundation development using SysGen requires a smaller design effort at the cost of performance and FPGA resource usage. This is because SysGen generates more code than a developer would have written by hand.

The first subsection gives an overview of SysGen. The second subsection describes hardware co-simulation. SystGen for DSP bears a strong resemblance to 20-sim. A comparison between the tools is given in the last subsection.

4.5.1 System Generator for DSP design flow

SysGen is a model-driven tool for configware development. The tool is build on top of MATLAB and Simulink, two tools which are widely used in the industry for solving all sorts of engineering problems. SysGen makes use of other Xilinx design tools and a finished design can be transferred to EDK and ISE Foundation.

Development of configware takes place using the graphical user interface of Simulink. SysGen is not able to generate code for standard Simulink blocks. For this reason Xilinx has added FPGA specific building blocks 'Xilinx blocks' to the library which are to be used for configware design. For simulation the Xilinx portion of a design can be interfaced to standard Simulink blocks. Interfacing of Xilinx and Simulink blocks requires 'Gateway' blocks to bridge the two different domains as illustrated in Figure 4.7. Gateway blocks perform sampling, datatype conversion and pin mapping. The Xilinx block domain is time-

discrete, uses a fixed time-step and uses a fixed-point datatype (the designer can customize the fixed-point format at the block level). Simulink offers additional timing simulations and offers more datatypes including integer and double precision floating-point.

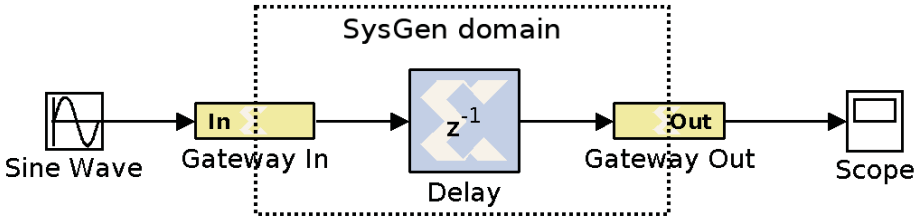

Figure 4.7: Design which mixes Simulink and Xilinx blocks

A Xilinx block is similar to a Simulink block except it has additional properties. Each block is backed by HDL-code and it has options for time delays, precision and rounding. Xilinx guarantees that if a design consists solely of Xilinx blocks simulations are cycle and bit-accurate. This means that the design will behave exactly the same in simulation as in an FPGA. Hence no structural or timing simulations are necessary.

SysGen provides a library of more than one hundred Xilink blocks. The library includes clones of common Simulink blocks like constants, adders and multipliers but it also includes FPGA specific blocks like registers, DSP blocks, communication busses and processor blocks.

The tool also offers the ability to express parts of a design in a subset of the MATLAB programming language which it can translate to HDL-code. This is useful for algorithms which do not lend themselves for expression in a graph.

A configware development flow for using SysGen in combination with EDK is shown in Figure 4.8. The figure is a realization of block [3] of Figure 4.4. In this section numbering in the form 3.b.x is used for substeps of [3].

The SysGen design flow assumes a finished design in Simulink consisting of a model and a test environment. Step by step, parts of the original Simulink model (step [3.b.1]) can be re-implemented using Xilinx blocks. Meanwhile the behavior of the design can be verified using the original MATLAB/Simulink test environment.

When the design is complete, SysGen can perform code generation. The tool offers various output options including HDL-code and a BIT-file. In case of a BIT-file, SysGen automatically carries out the Design Synthesis [3.b.2] and Design Implementation [3.b.3] steps using ISE Foundation. Optionally the user can decide to carry out structural and timing simulations but as mentioned before, these steps are not necessary if the design consists solely of Xilinx blocks since Xilinx guarantees bit and cycle accuracy.

SysGen cannot predict beforehand whether the design meets all timing constraints. For this reason step [3.b.3] might fail and changes to the design might be required e.g. the addition of registers for adding a pipeline if the design contains long combinational paths.

The end-result of step [3.b.3] is a BIT-file which can be tested on the FPGA in step [3.b.4]. If the design proves to work correctly for example when connected to a plant, the SysGen project can be exported to the EDK [3.b.5].

The default CPU used for designs on the ML510 is the built-in hard-core PowerPC 440. Xilinx also provides a soft-core CPU named MicroBlaze. The MicroBlaze can be selected in the EDK as a general purpose processor like the PowerPC 440. The design flow as

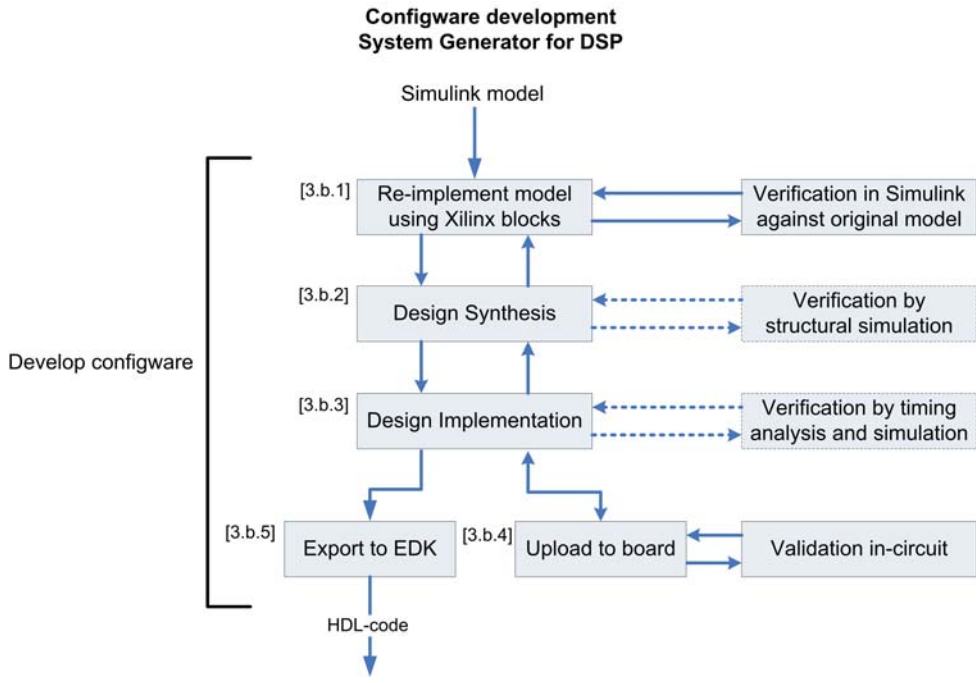

Figure 4.8: System Generator for DSP configware development flow

described in this section can be followed for it as well, but two additional flows are specially available for the MicroBlaze:

1 Import MicroBlaze EDK design into SysGen
An EDK design consisting of MicroBlaze and peripherals like a Uart and Ethernet can be imported into SysGen. The design can then be extended in SysGen with Xilinx blocks and SysGen is responsible for generating a BIT-file of the complete design. This functionality allows someone without EDK experience to create a configware design.

2 MicroBlaze Xilinx block
The MicroBlaze is available as a special Xilinx block in Simulink. The block allows the inclusion of a MicroBlaze into a SysGen design without having to use EDK for setting up a design. A software executable in the form of an ELF-binary can can be set as an option in the block.

SysGen is not the only MATLAB/Simulink to HDL-code tool on the market. The MathWorks, the company behind MATLAB and Simulink, also offers an HDL-addon for Simulink called 'Simulink HDL Coder' (The Mathworks, 2009). The tool has similarities with SysGen as both can generate HDL-code. The main difference is that HDL Coder can generate code of a normal Simulink model while SysGen is limited to special Xilinx provided blocks. The functionality of HDL Coder is is limited to code-generation and it exposes no FPGA-specific blocks for DSP blocks, processor busses or memory. Further external HDL design tools are needed for synthesis and 'FPGA-in-the-loop' simulations.

4.5.2 Hardware co-simulation

System Generator for DSP provides support for hardware co-simulation which is similar to FPGA-in-the-loop, see Figure 4.3. In a hardware co-simulation, the parts of a design implemented using Xilinx blocks in Simulink are run on the FPGA and the Simulink parts (the testbench) are executed on the development PC. Communication between FPGA and development PC takes place using JTAG or Ethernet. The latter provides more bandwidth

and is that way suited for high bandwidth applications or a high simulation frequency. JTAG is available on all boards but Ethernet is not available on all and is also not support in the Linux version of SysGen (version 11.1),

A hardware co-simulation is set up by selecting 'Hardware co-simulation' as the compilation target in SysGen. During compilation SysGen creates a BIT-file, a Simulink wrapper block and Matlab scripts. The BIT-file is synthesized from two sources of HDL-code namely HDL-code generated from the design and wrapper HDL-code. The wrapper HDL-code forms the communication bus at the FPGA side between the ML510 and Simulink. At the development PC, a Simulink wrapper block acts as a gateway from Simulink to the FPGA. Next to the Simulink wrapper block also MATLAB scripts are generated. These allow hardware access from MATLAB without any involvement from Simulink. The choice for using either MATLAB or Simulink depends on the type of tests the designer wants to perform. A reason for using MATLAB, can be that in a script the designer has more control over the provided test signals and data logging.

Two types of simulation are supported: single-step and free-running. When using single-step, the FPGA and simulation clocks are synchronized. This allows for bit- and cycle-accurate simulations but synchronization overhead limits performance.

In free-running mode, FPGA and simulator clocks are not synchronized. Both simulator and FPGA can work as fast as they can which results in higher simulation performance. When an event occurs in simulation, SysGen communicates with the FPGA. Because the clocks are not synchronized, an unknown number of FPGA clock cycles has elapsed before the data has reached the FPGA. Due to this, the simulation is not bit- and cycle-accurate. The user needs to provide his own synchronization mechanism.

In general, a hardware co-simulation using SysGen is an FPGA-in-the-loop simulation. The tool also offers the ability to include external signals e.g. from AD-converters into a hardware co-simulation. This way a hardware-in-the-loop simulation can be performed. The SysGen option for this is not clearly visible in the tool if one wants to use it look for 'non-memory-mapped ports'.

Groothuis (Groothuis, 2004) designed a hardware-in-the-loop simulation setup consisting of a real embedded control system and a real-time simulation model of a plant. The embedded control system consisted of a PC/104 which ran the control algorithm on real-time Linux. Another PC/104 system ran the real-time simulation model of the plant. The output of the control system were PWM signals and the system which ran the simulation required PWM inputs. The difference with Groothuis his approach is that in this case the control algorithm runs on an FPGA instead of a CPU. Further setting up the simulation in SysGen takes a smaller design effort because big parts of the work are automated by the tool. One of the things Groothuis had to do was to provide his own communication bus.

4.5.3 Comparison between Simulink/System Generator for DSP and 20-sim/4C

Simulink/System Generator for DSP and 20-sim are both model-driven design tools but each have been designed for a different purpose. In this subsection the tools will be compared on code-generation and interfacing the generated code to physical hardware.

System Generator for DSP is a generic configware design tool which can be used for all sorts of signal processing tasks. Modelling takes place using blocks which are interconnected by signals. During code-generation each block is replaced by a HDL-code template and all these blocks are connected by a HDL-code wrapper.

20-sim has been designed for mechatronic modelling and design. A mechatronic design

consists of a plant and a controller. The physical model of the plant is modelled using equations, iconic diagrams or bond graphs and these models use power ports or signals. A digital controller works using signals. 20-sim code generation is based on templates which contain placeholders for equations and design specific information. Before code-generation takes place, 20-sim converts the model to an ordered set of equations. During this process for example algebraic loops are solved by solving the respective equations. When generating code, the set of equations is converted to sequential code in e.g. C or C++ and placed in the template.

Code-generation at the block level as performed by SysGen has two advantages. First of all an FPGA is inherently a parallel device. If the SysGen design contains parallel signal paths, parallel HDL-code is generated which is also synthesized to a parallel structure on the FPGA, so there is a close mapping of the design onto hardware. Secondly, each block can be backed with highly optimized HDL-code because blocks are independent. A drawback is that SysGen does not handle algebraic loops well. If it encounters one during simulation or code-generation, it warns the user about it by mentioning that results do not converge and it suggests to add delays to fix the problem. 20-sim attempts to solve algebraic loops when it encounters them. The structure of the code generated by 20-sim is not a direct mapping of the structure of the model on code. For a general purpose processor 20-sim generates sequential code and this code is less suited for mapping to an FPGA because the parallelism of the model is lost in the code.

A design in SysGen is time-discrete and is limited to fixed-point. 20-sim can be used for the modelling of both continuous-time and discrete-time systems and it uses double precision floating-point internally. This format is more suited for control than fixed-point because it is able to represent small numbers with more accuracy than fixed-point.

After code-generation, the generated code needs to be extended with platform specific code for interfacing to I/O pins. In case of 20-sim this task is handled using 4C which is an external tool. SysGen offers two ways for performing pin mapping. In the first method pin mapping is performed internally inside the 'Gateway blocks' of the SysGen model, so design and pin mapping are not separated. In the second method the SysGen design is exported to EDK. Pin mapping has to be performed in the EDK inside a UCF-file. The EDK could be considered similar to 4C for this task.

The features of both tools are summarized in Table 4.1.

	Tool	
Feature	System Generator for DSP	20-sim
Code generation	block-level	model-level
Code structure	sequential, parallel	sequential
Timing model	discrete-time	continous-time, discrete-time
Floating-point type	fixed-point	double-precision
Pin mapping	in model or outside using EDK	outside model using 4C

Table 4.1: Comparison of System Generator for DSP and 20-sim/4C features

Both SysGen and 20-sim use a different method of code generation where SysGen is optimized for parallel execution on FPGAs and 20-sim for sequential execution on a CPU. If 20-sim could generate sequential HDL-code instead of C/C++, SysGen and 20-sim could be considered opposites in the method of code generation. Depending on the situation one of the methods is better than the other. Under FPGA resource constraints sequential code could be better because it allows for re-use of LUTs or DSP blocks for computations. A

parallel implementation has a close mapping to physical hardware and makes debugging easier because there is a physical FPGA net for each input or output.

5 Proposed design flow

Section 1.3 describes the CE design flow, used for realizing a design on an embedded hardware platform. It is explained that 20-sim does not provide a code-generator for HDL. This means that for realizing a design in configware on the ML510, external configware design tools must be used. In this chapter the CE design flow is adopted with a Xilinx hardware/software co-design flow.

The first section, considers the realization of a 20-sim controller design on the ML510 a 'Hardware/Software co-design' problem. It describes the partitioning of the controller design in software and configware. The second and third section describe which tools should be used for the development of respectively configware and software in context of the modified CE design flow.

5.1 Partitioning of a controller design in software and configware

During the Realization phase of the CE design flow (see Figure 1.2), a 20-sim controller design is transferred to an embedded hardware platform. This process can be considered a 'Hardware/Software co-design' problem for which a partitioning in software and configware has to be made.

In this section, the design of a 20-sim controller is partitioned in software and configware for use on the ML510. In order to establish this partitioning, two partitionings used in previous projects are discussed and the results are used for the new partitioning.

The first subsection describes a partitioning in which the control algorithm is implemented completely in software. The second subsection describes a partitioning in which the control algorithm is implemented fully in configware. The proposed partitioning, as described in the third subsection, uses a combination of both software and configware to come to a better balance between software and configware.

5.1.1 Software based control implementation

In most robotic and mechatronic projects at CE a demonstrator is built to prove a certain theory. The focus lies on the theory and not on the creation of the demonstrator. A demonstrator consists of a plant and a controler to control the plant. The controller is implemented on an embedded hardware platform which consists of an embedded CPU in combination with a small FPGA. A fixed partitioning in software and configware (as shown in Figure 5.1) is used.

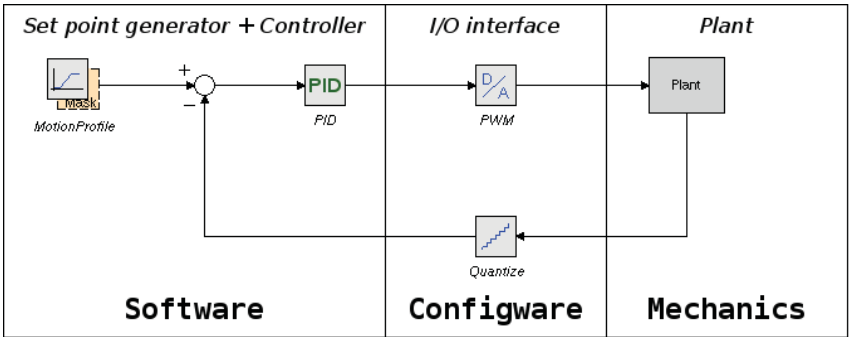

Figure 5.1: Partitioning in software/configware used in most CE projects

The figure shows three blocks: software, configware and mechanics. The software and

configware blocks are discussed here. The software block contains a set point generator and a controller. Code for these parts is generated by the 20-sim C/C++ code generator. The configware block contains pre-built I/O interfacing blocks for digital-to-analog and analog-to-digital conversion. Using the 4C tool, the input/output ports of the generated C/C++ are interfaced to these I/O interfacing blocks. Next, 4C compiles the C/C++ code to an executable using cross-compilers created by Embryo. The executable is run on an embedded Linux distribution created by Embryo.

To summarize: the amount of design effort needed for porting a controller design to an embedded hardware using this partitioning is limited, because the software is generated automatically and for configware pre-built I/O blocks are used.

5.1.2 Configware based control implementation

A controller design can also be implemented fully in configware as illustrated in Figure 5.2. This partitioning has been used for the Production Cell (see section 1.1 and 2.5). The Production Cell implementation worked correctly, but a large amount of design effort was required in order to fit all parts into an FPGA.

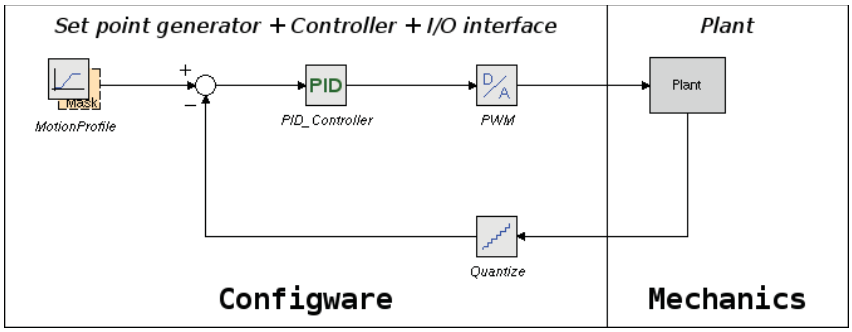

Figure 5.2: Partitioning in which the design is implemented fully in configware

5.1.3 Proposed partitioning

Due to tight integration of FPGA logic and PowerPC CPUs, different parititionings in software and configware are possible on the ML510. A controller design does not have to be implemented completely in software or completely in configware anymore. A combination of both can be used in which there is a better balance between software and configware.

The choice, how parts of the controller design must be implemented, depends on the requirements defined for the application. In general the strengths of each technology should be exploited. In case of software three situations can be identified in which it should be used. First of all software should be used for tasks with low real-time requirements. Second, software should be used for tasks which require flexibility in updating the algorithm. Third, software can best be used for floating-point calculations because in configware these require a large amount of FPGA resources in configware, but it should only be used for this if real-time requirements can be met. Configware should be used for tasks with hard real-time requirements, tasks which are computation intensive and for I/O.

In case of a controller design with a structure similar to Figure 5.1, a better balance between software and configware can be achieved by implementing the PID-controller in configware, so that hard real-time behavior can be assured. The set point generator, which typically uses floating-point, can be implemented in software (see Figure 5.3) by filling a block of memory which is accessible by the PID-controller. This way the design is more flexible compared to the complete configware implementation because software is easy to

adjust, so a differently shaped motion profile can easily be created. If the CPU initializes the block memory before the controller is started, real-time behavior can be guaranteed without having to use a real-time operating system.

During design, co-simulation must be used to verify correct behavior. Damstra (2008) describes how a co-simulation between different design tools can be set up for mechatronic design. In previous projects co-simulation was not used because it was not needed or not possible. Co-simulation is not needed for a pure software implementation of the controller because all models are in the same tool. For a setup like the Production Cell, it is needed because configware and the plant model are not in the same tool. In case of the Production Cell, co-simulation has not been used because the configware design tool offered limited simulation support. Instead the behavior was validated on the setup.

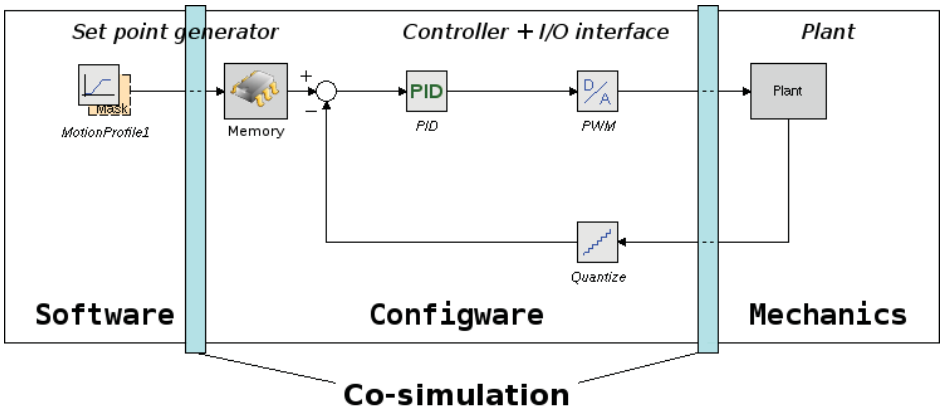

Figure 5.3: Partitioning in software/configware optimized for the ML510

5.2 Configware development

After a design has been partitioned in software and configware, the configware portion can best be implemented using SysGen. This tool has been chosen for two reasons. First of all, the use of SysGen requires only a limited amount of computer engineering knowledge. Second, SysGen can be used for model based design which is comparable to development in 20-sim. A design created using SysGen might have a higher FPGA resource usage and lower performance than a design developed using manually written HDL-code but the size and clock frequency of the Virtex-5 compensate for that.

The process of porting a 20-sim design to an FPGA using SysGen is illustrated in Figure 5.4. The figure has been derived from Figure 4.8. The first two steps are different, the rest of the flow is the same and has been shown for completeness.

- Separate model into software, configware and mechanics [3.c.1]
  The input to the design flow is a completed 20-sim model of the controller. The model should be separated into the parts to be implemented in software, configware and mechanics as indicated in Figure 5.3. This is needed to prepare the model for a co-simulation of configware and mechanics.
- Re-implement Controller and I/O using Xilinx blocks [3.c.2]
  The 20-sim models of controller and I/O are re-implemented in Simulink using Xilinx blocks. The behavior of controller and I/O should be verified in a co-simulation against the model of the plant in 20-sim. This comparison can for instance be used to compare the accuracy of double precision floating-point (as used by 20-sim and Simulink) to fixed-point (as used by SysGen).
- Configware flow [3.c.3 - 3.c.6]

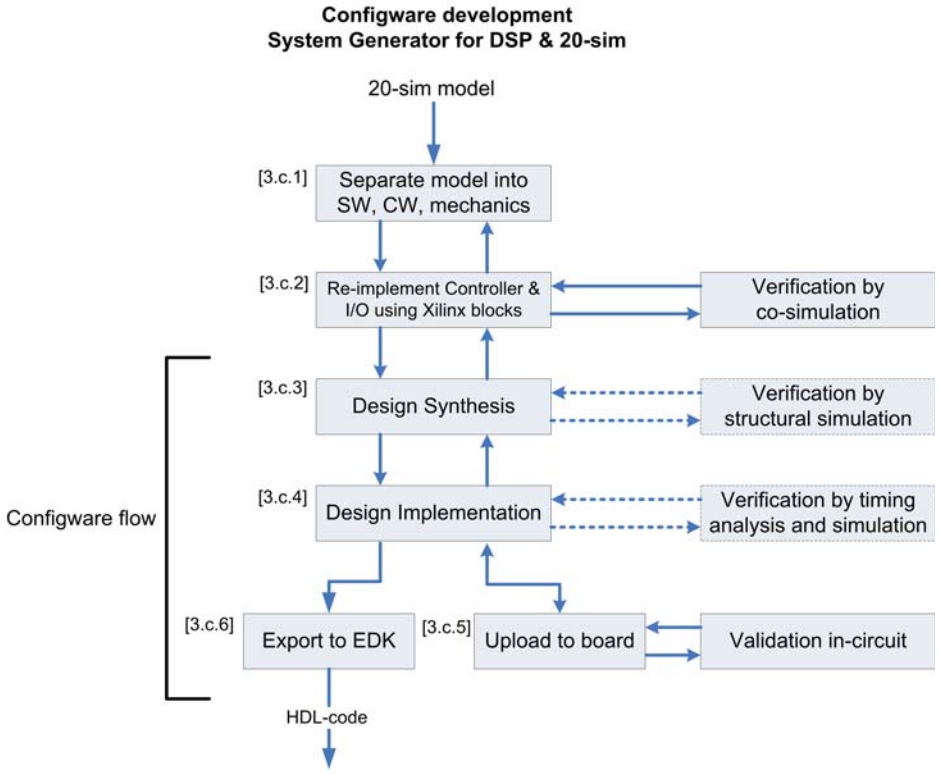

Figure 5.4: Proposed design flow using 20-sim in combination with SysGen

The standard SysGen configware design flow is resumed at the 'Design Synthesis' phase. As has been described before, simulations can be carried out at each stage of the configware design flow. Xilinx recommends to skip simulation at the Design Synthesis and Design Implementation stages if the design is composed solely of Xilinx blocks, because of guarantees for cycle and bit accuracy which they provide for Xilinx blocks. The end-result of the design flow is HDL-code which can be imported into Platform Studio EDK for inclusion in the system design.

The design flow actually used in this assignment is slightly different from the described flow. Instead of setting up co-simulation between Simulink and 20-sim, the 20-sim model of the plant was exported to a Simulink model in order to not have to set up a co-simulation environment which is time consuming. Using this mechanism the Controller could also be brought over to Simulink for direct comparison.

5.3 Software development

As part of this assignment, support for the ML510 has been added to Embryo. This section describes integration of Embryo into the Xilinx software development flow.

Xilinx recommends the use of Platform Studio SDK for the creation of a BSP and software development. Unfortunately the functionality of Platform Studio SDK is limited to BSP creation when the target operating system is not one of Xilinx their own operating systems. A different tool for example the standard version of Eclipse should be used for software development.

Embryo has built-in support for the ML510, but for building the Linux kernel external information in the form of a *Device Tree Source* file (DTS) is needed. This DTS-file provides the Linux kernel with a list of configured peripherals and for each peripheral

its memory address offset and interrupt. For most standard hardware platforms this information is fixed or can be obtained at system powerup from firmware. In case of the ML510 the information is specific to a BSB.

A BSP-generator plug-in for Platform Studio SDK named 'Device-tree' (Xilinx, 2009b) is used for the creation of a DTS-file. Cross-compilers generated by Embryo are used for compilation of the Linux kernel and software.

The software development half of Figure 4.4 extended with Embryo is shown in Figure 5.5. The numbering used in the figure does not reflect the order in which steps are performed. Previously the 'Upload to CompactFlash' step was placed under the EDK. In the Embryo flow there are two partitions on the CompactFlash card one for the ACE-file and one for the Linux root file system and the Linux ELF-binary, therefore [2.b.6] has been placed in the software development flow.

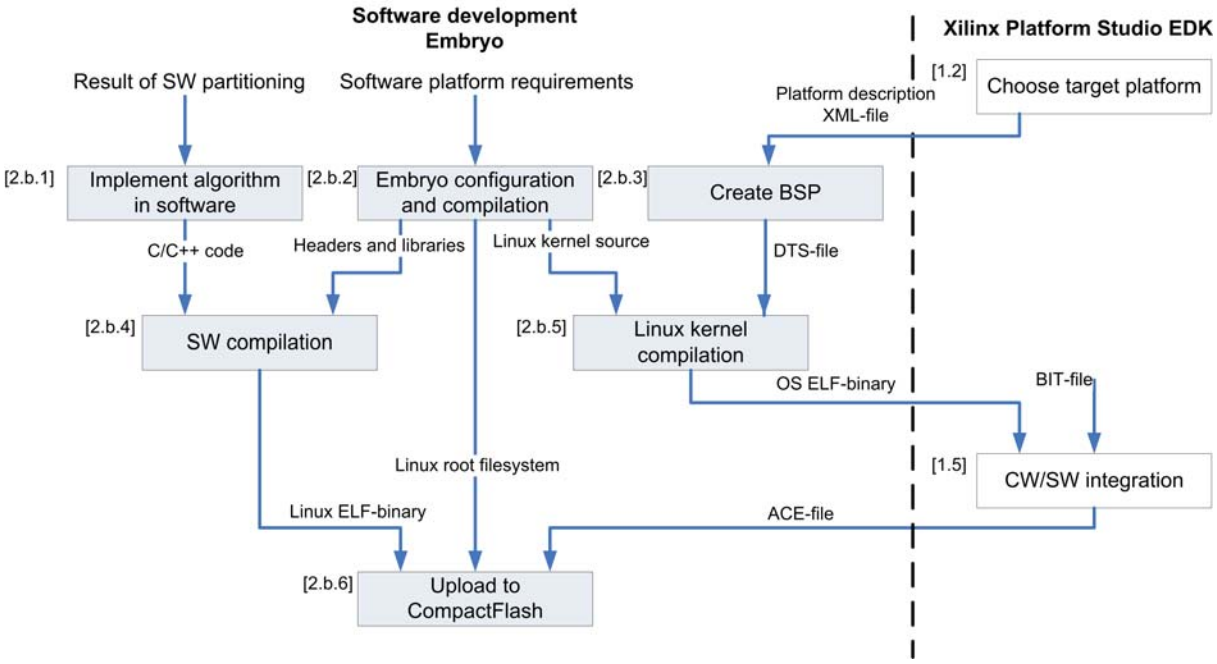

Figure 5.5: Embryo flow

The steps of the design flow are described below:

- Implement algorithm [2.b.1]
  The software parts of the algorithm are implemented in C or C++.
- Embryo configuration and compilation [2.b.2]
  The configuration of Embryo starts with a list of requirements (e.g. required libraries and tools) for the software platform. Based on these requirements Embryo is configured and after that compiled. This step results in development files (headers and libraries) to be used during cross-compilation of the software algorithm, a Linux root filesystem and a configured Linux kernel source tree.
- Create BSP [2.b.3]
  The Device-tree BSP-generator is used for the creation of a BSP.
- SW Compilation [2.b.4]
  C/C++ code is compiled using the cross-compilers created by Embryo.
- Linux kernel compilation [2.b.5]
  During Linux kernel compilation, a configured Linux kernel source tree is combined with the DTS-file for the creation of an ELF-binary containing the Linux kernel.

- Upload to CompactFlash [2.b.6]
  In the final stage the ACE-file is written to a FAT-partition on a CompactFlash card. The Linux root filesystem and Linux ELF-binary of the algorithm are both written to a Linux-partition on the same CompactFlash card.

This section described integration of Embryo with the Xilinx software flow. More information on how to compile Linux for the ML510 using Embryo can be found in **??**.

6 Design of a robotic demonstrator

The previous chapter describes a design flow for implementing a controller design in the Virtex-5 FPGA. In this chapter, this design flow is applied on the design of a robotic demonstrator in which a motor is controlled by the Virtex-5. The first section describes the demonstrator setup. The second section describes the FPGA implementation of the controller. The third section describes the results.

6.1 Description of demonstrator setup

A robotics demo setup developed within CE is the JIWY as shown in Figure 6.1. The JIWY is a mechatronic setup for holding a camera. Two motors can rotate the setup in respectively the horizontal and vertical direction.

This project focuses on proving that the ML510 is suitable for control applications. Because the JIWY is used in a lot of different projects, it has been modelled already and for this reason is used as the demonstrator setup for this project as well. In this project only the horizontal axis of the JIWY is controlled by the Virtex-5. The remaining text of this section describes more details about the JIWY setup relevant for the demonstrator.

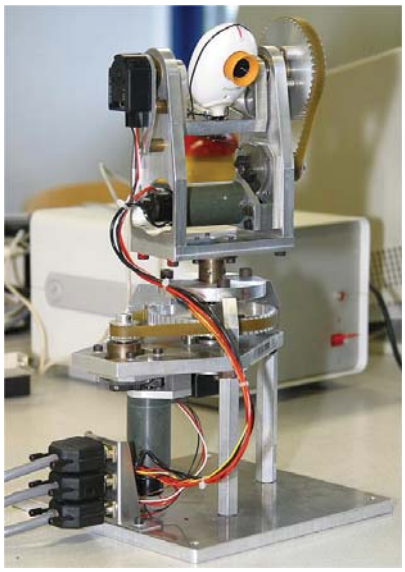

Figure 6.1: Robotics demo setup

The horizontal platform of the JIWY is rotated by a DC motor which is driven by a PWM signal. There are two endstops on the platform which prevent the platform from making a full rotation. The angular position of the platform is measured by a quadrature encoder which is mounted on the horizontal axis. The output of the encoder are pulses which need to be counted in order to obtain the position. In case of the JIWY, one rotation corresponds to 2000 counts.

In this project the model of plant and controller have been reused from (Deen, 2008). The plant model has been used without any modifications but changes have been made to the controller. The original controller consisted of a basic proportional controller and it has been changed to a PID controller. The model of the PID controller to be implemented in the Virtex-5, is shown in Figure 6.2. The controller has two inputs for the set point and

the encoder. Both inputs provide an angle in encoder pulses for which 0 pulses corresponds to 0 degrees and 2000 pulses to 2π rad.

When a controller is designed during the Control Law Design phase, see Figure 1.2, it is designed to work on signals which have a value and a dimension like radian, Volt or Newton. In the hardware implementation of the controller, signals are represented by digital numbers which are typically integers and which have a different value range than the original controller. A design choice has to be made whether to scale the numbers to lie in the same value range as the orignal controller design or to modify the controller to work on the larger value range. In case of scaling, floating-point is required because the numbers become small. When the controller design is modified to work on the value range of the measurement data, the hardware implementation is not a direct mapping of the original controller design on hardware.

In case of the Jiwy encoders it has been decided to convert encoder pulses to radians, so that no modifications to the controller design were needed. This conversion requires a conversion factor of $\pi/1000$ from encoder pulses to radians. The output of the controller, Controller_OUT, is connected to a PWM-generator. The range of Controller_OUT is limited from -1 to +1 because these values are mapped to PWM in which -1 and +1 corresponds to respectively rotating full speed to the left and full speed to the right.

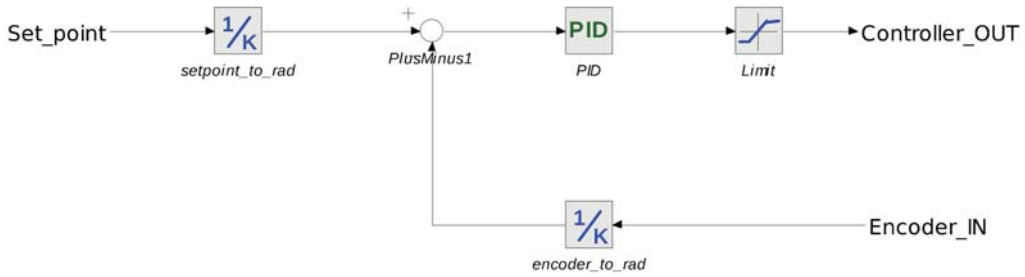

Figure 6.2: 20-sim model of position controller

The equations for the PID algorithm as used in the 20-sim model are:

```
factor = 1 / ( sampletime + tauD * beta );
uD = factor * ( tauD * previous ( uD ) * beta
     + tauD * kp * ( error - previous ( error ))
     + sampletime * kp * error );
uI = previous( uI ) +  sampletime * uD / tauI;
output = uI + uD;
```

A 20-sim simulation showing the step response of the controller connected to a model of the JIWY is shown in in Figure 6.3. The controller is simulated at a sample frequency of 1kHz and the parameters used for the PID algorithm are: Kp=10, tauD = 0.02s, beta=0.1, tauI=10s.

As can be seen in the figure it takes approximately 0.1 second for the JIWY to make a rotation of $\frac{1}{2}\pi$ rad.

6.2 FPGA implementation of position controller

This section describes the implementation of the position controller in the Virtex-5 FPGA. The first subsection describes the overall design. The remaining subsections describe the overall design.

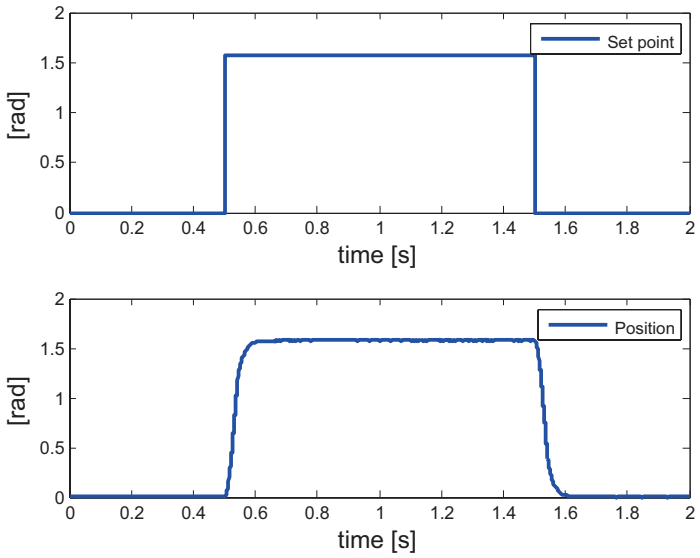

Figure 6.3: Step response of position controller in 20-sim.

6.2.1 Design

In subsection 5.1.3 a partitioning of a controller design is made in software and configware (see Figure 5.3). For the JIWY position controller the same partitioning is used. The design of the configware part made using SysGen is shown in Figure 6.4. The figure is based on Figure 6.2 but has been extended with a PWM generator, an encoder interface and a processor interface. The set point and measured value are in integers and are subtracted in this format as well because an integer subtraction is more efficient in terms of FPGA resource usage. The error value is converted to radians inside the PID controller block.

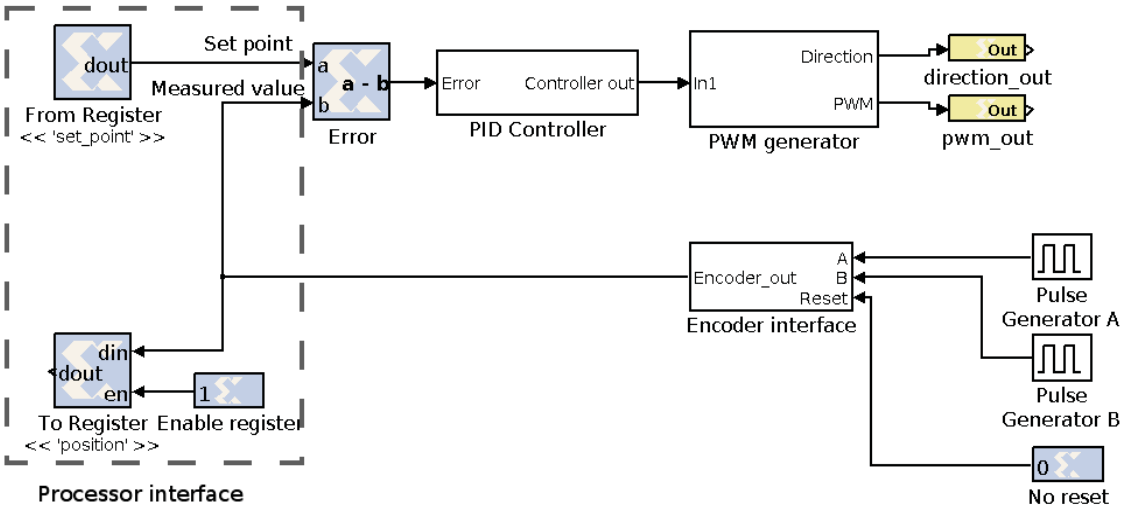

Figure 6.4: PID-controller implemented using Xilinx bloxks im Simulink

The design consists of five blocks:

- Processor interface
  The processor interface provides read and write access from a PowerPC 440 core to the SysGen design using the PLB bus. This interface is used for providing a set point value and for read back of the horizontal position from software. The position value could be used for a homing procedure at startup.
- Error calculation
  An error value is calculated by taking the difference between the set point value and the measured value. The result is passed to the PID controller.
- PID controller
  The PID controller calculates a control value based on the error value and internal states variables.
- PWM generator
  The PWM generator generates a PWM signal and a direction signal based on the control value.
- Encoder interface
  Pulses A and B, generated by the quadrature encoder are counted, in order to calculate the horizontal position of the JIWY setup.

The set point generator is implemented in software. A Linux program was written which accesses the 'set_point' register of the processor interface using memory mapped I/O.

6.2.2 PID controller

This subsection describes the implementation of the 20-sim PID algorithm in SysGen. An image of the complete design is placed in Appendix C but it is not needed for understanding this subsection.

As described previously, the main goal for this demo is to prove that the ML510 is usable for application in robotics. For this reason the design of the PID algorithm has been kept simple and no optimizations have been added.

Each operation of the 20-sim PID algorithm has been mapped directly on a Xilinx block. Designs in SysGen must use fixed-point and so must this design. Based on predictions of the signal levels inside the algorithm, a fixed-point format has been chosen for each block. In general a 32-bit fixed-point format with 24-bit fraction bits has been used. The number of integer bits has been increased at the cost of fraction bits in places where the numbers can become large.

The design does not make explicit use of DSP blocks for which either special Xilinx blocks have to be added to the design or special options need to be set on Xilinx blocks. The synthesizes tool may decide to use DSP blocks depending on constraints but for this design it has not used them.

During design, the behavior of the SysGen implementation of the PID algorithm has been verified against a 20-sim model of the JIWY. The JIWY model was exported to Simulink and imported into the SysGen design using the simulation setup shown in Figure 6.5. Because 20-sim is not able to export models of I/O interfacing blocks like analog-to-digital converters, these have been implemented using Simulink blocks. The behavior of the Simulink equivalents of the 20-sim blocks can differ slightly.

The result of a step signal applied to the SysGen version of the position controller is shown in the top of Figure 6.6. The same test signals have been applied as used in Figure 6.3. The bottom of the figure shows the difference between the same simulation carried out using 20-sim. As can be seen upto 1.4s the maximum error is 1 count and at approximately 1.5 seconds the maximum error between 20-sim and SysGen is 5 counts which is equivalent to 1 degree. This difference has three causes. First of all the 20-sim model uses double-

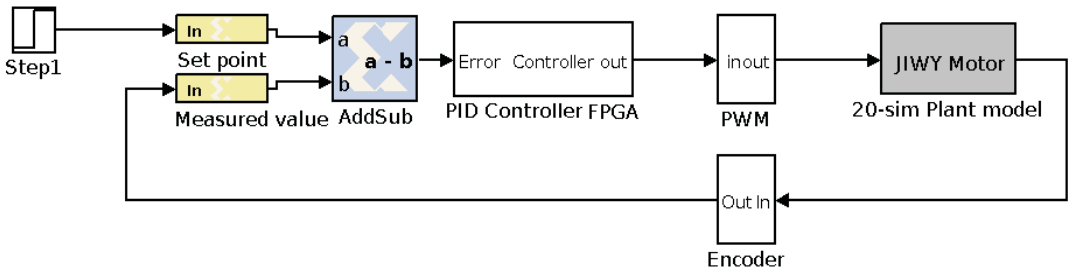

Figure 6.5: Simulation setup used for verification of the controller design against a 20-sim model of the JIWY

precision floating-point while the SysGen implementation uses fixed-point because 20-sim does not support fixed-point. Second, each simulation tool uses a different numerical integration method. Third, different models for I/O blocks have been used in Simulink and 20-sim which also behave differently.

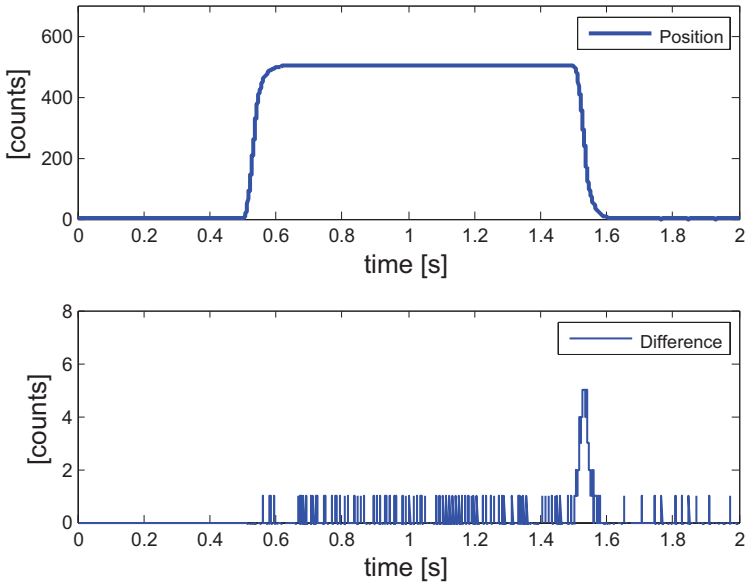

Figure 6.6: Step response of position controller implemtented using Xilinx blocks

6.2.3 Encoder interface

The quadrature encoder mounted on the horizontal axis of the JIWY is connected to the encoder interface. As explained before, a quadrature encoder outputs two pulse signals which correspond to the relative horizontal displacement. These pulses need to be counted in order to obtain the position. The algorithm for counting pulses can be implemented as a graph in SysGen but it was decided to implement it in MATLAB code. The reason for this is that the algorithm is easier to express in code and further it allows for testing MATLAB to HDL-code conversion by SysGen. This subsection gives a description of the MATLAB code and ends with a simulation result.

As shown in Figure 6.4, the encoder interface has three input signals (A, B and reset)

and one output containing containing the number of pulses. The inputs correspond to the parameters passed to the MATLAB function and the output is the result of the MATLAB function:

```
function [Encoder_Out] = encoder_func(A, B, Reset)
```

Internally the algorithm contains state variables for the pulse count and the previous values of A and B. Normally variables in MATLAB are typeless until they are assigned a variable. For conversion to HDL-code it is required that datatypes are known in advance. The following code defines the pulse count variable as a 12-bit signed register:

```
persistent count, count = xl_state(0, {xlSigned, 12, 0});
```

The encoder pulses A and B are square waves which are 90 degrees out of phase. Depending on the direction in which the JIWY setup rotates, the A signals leads the B signal or the way around. Depending on the current and last values of A and B, the pulse count is increased or decreased when a rising or falling edge of one of the signals occurs. The MATLAB sample code, which is a small part of the total counting algorithm, illustrates pulse counting for a situation in which A leads B and a rising edge of A occurs:

```
%if rising edge of A and B=0
if(A==1 && A_previous==0 && B==0 && B_previous==0)
    count = count + 1;
```

A simulation of the complete encoder interface is shown in Figure 6.7. The top graph in the figure shows two pulses A and B which are 90 degrees out of phase. The bottom graph shows the pulse count which is increased when a rising or falling edge of one of the signals occurs.

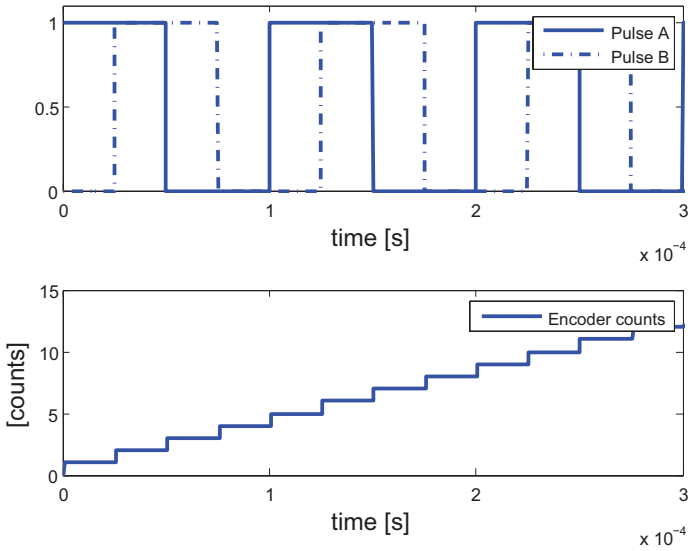

Figure 6.7: Counting of encoder pulses

The encoder interface has been tested separately of the rest of the design on hardware. For this the highest bits of the encoder output where interfaced to the LCD interface of the ML510. These pins were connected to an oscilloscope with digital inputs.

During testing it appeared that the counting algorithm sometimes does not reach the same pulse count when rotating the JIWY back and forth between the start position. The cause for this was found in the low analog bandwidth of the optocouplers and the voltage levels used on the ML510 I/O board. The input signals to the optocouplers are pulses with steep edges but after the optocouplers the edges are not steep anymore and the digital high level after the optocouplers is 1.8V while the digital threshold voltage of the Virtex-5 inputs is 1.25V. Filtering was added to the code to compensate for the limitations of the ML510 I/O board and with the filtering code the pulse count is the same each time when rotating the JIWY back and forth between the start position.

6.2.4 PWM generator

PWM signals are oftenly used for digital-to-analog conversion in digital control systems. In this case the motor used for horizontal positioning of the JIWY setup is actuated using a PWM signal.

The PWM generator creates a PWM signal and a direction signal for driving an H-bridge to which a motor is connected. The PID controller generates an output value between -1 and +1 which correspond to respectively rotating to the left and to the right. The output value of the PID controller is scaled to an integer between -10000 and +10000 which is used as a threshold value. The PWM generator generates a signal by comparing the absolute value of its input value against a sawtooth signal generated by a counter. When the input value is below the counter value, an output of 1 is generated else 0. Depending on the sign of the input value the direction signal is made high or low.

A simulation of the PWM generator is shown in Figure 6.8 for a fixed input value. The counter counts from 0 to 10000 and is compared with a fixed threshold value of 2500 which results in a signal which is high for 25% of the time.

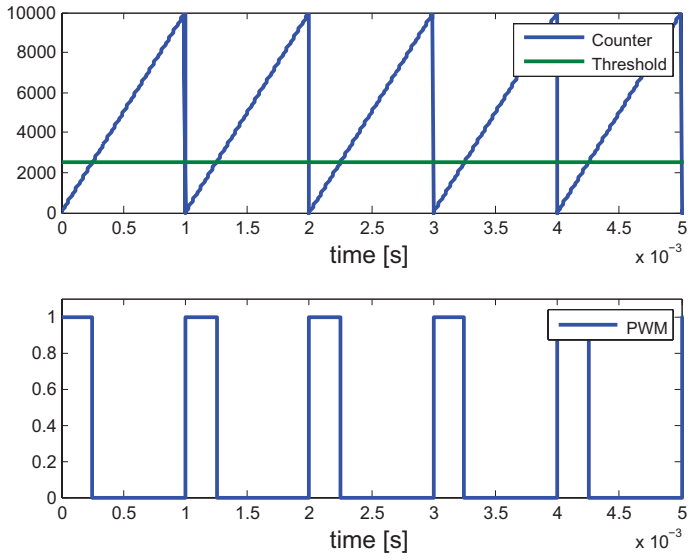

Figure 6.8: Simulation of the PWM generator

6.3 Results

In this chapter the design flow proposed in chapter 5 was applied to the creation of a robotic demo setup. The first subsection the results of using the design flow. The second subsection describes the results of the controller design.

6.3.1 Design flow

As explained in section 4.2, during the first steps a base system is created which forms the foundation of an embedded design. This wizard proved to be easy in use and within an hour you obtain a synthesized BIT-file.

Getting Linux to work, using this BIT-file, requires extracting a DTS-file and combining this with the Linux kernel sources. The device-tree generator works well for generating a DTS-file. After compilation of the Linux kernel with a correct DTS-file, Linux should boot on the ML510. It should then behave like any other Embryo installation and anyone with Linux experience should be able to use it.

For configware development SysGen proved to be an easy to use tool but it has some issues. As claimed by Xilinx, it is possible to create functioning configware designs in SysGen with limited computer engineering knowledge. After creating a design only a few clicks are required to have SysGen generate a BIT-file. This functionality has been used for testing the encoder interface and the PWM generator on hardware without a complete processor design. It takes approximately five minutes to synthesize a small design and upload it to the Virtex-5 using JTAG. This allows for short design iterations.

When the design becomes more complicated, synthesis can fail during the final stages when applying timing constraints for I/O or bus interfaces. At such moments computer engineering knowledge is still needed to fix the problem. In most cases the problem is that the design contains so called 'long combinational paths'. This are paths consisting of a large number of logic gates without any sequential element like a flip-flop in between. Long combinational paths can introduce long timing delays and due to these timing constrains can not always be met. Useally this problem can be fixed by adding delay blocks. If the PID controller (as shown in Appendix C) would not meet timing constraints, a redesign is required. The reason for this is that the design itself runs at 1kHz, so each delay block would add a delay of 1ms which would change the behavior of the PID algorithm.

If the complete configware design needs to be used in combination with a CPU, it must be exported to the EDK. During import of a configware design, made using either SysGen or ISE, in EDK, the EDK import tool recognizes bus connections and even connects the design to the PLB bus. After that the designer needs to perform some steps by hand: assigning the memory address, interfacing of I/O ports to FPGA pins and specifying the timing constraints. The main problem encountered during this project when integration a design into EDK were again violations of timing constraints. In theory this issue can be fixed in the same way as described above. The problem is that the issue is more complicated to find as the violation might occur in other peripherals. The exact cause can be found by an analysis of the (generated) HDL-code and timing logs.

In the end, integration of the controller design into the EDK succeeded. The controller design could be accessed from software by accessing the memory addresses configured in the EDK. For this a small Linux program was written for specifying a set point and reading of encoder values.

6.3.2 Controller design

The Jiwy position controller design works in configware and software as illustrated by a measurement shown on the actual setup as shown in Figure 6.9. The same set point

was applied to the Jiwy setup as in the simulation. As shown in the figure the measured position suffers of 20% overshoot and it takes more time for the setup to reach the set point for the first time. The exact causes of this issue have not been investigated. One cause is that the PWM frequency is 1kHz while the motors need a frequency between 10kHz-20kHz for accuracy. This is due to a design error which was revealed during testing where the complete design worked 10 times too fast (PWM worked at 10kHz at this high frequency). The clock frequency of the design was reduced from 100MHz to 10MHz to get it running at the correct speed and this reduced the PWM frequency to 1kHz as well. There was no time to fix this. A second cause is that the plant model of the JIWY is relatively basic and does not take into account all friction effects of the setup. The Jiwy setup also does not reach its final value of 500 but reaches 491 counts which is 1.6 degrees off. The cause of this is that there is a deadzone upto approximately 10% in the PWM output, which is caused by the low analog bandwidth of the optocouplers on the ML510 I/O board. Compensation could be added for the deadzone but this has not been done.

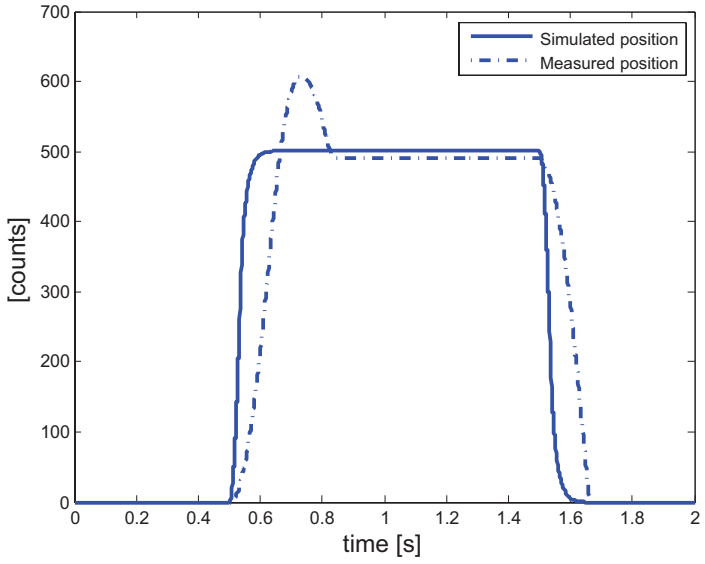

Figure 6.9: Position measurement of Jiwy setup compared to simulation

The FPGA resource usage of the controller design is shown in Table 6.1. The table shows the FPGA resource usage of the blocks, see Figure 6.4, in slices, flip-flops and LUTs. This information comes directly from the synthesis logs but the slice consumption needs some additional explanation. As explained previously a slice is made up of four LUTs, four flip-flops and some other logic. Parts of a slice can be used by multiple blocks. This results in the synthesis tool counting some slices multiple times. Due to this the total slice usage, in this case 700, is lower than the sum of the slice count per block.

In total the controller design requires 700 slices which corresponds to 3.4% of all slices. To get a feeling whether the SysGen design is small or large, an attempt is made to compare the results to an integer implementation of the Production Cell. It cannot be compared to the floating-point version because that implementation is too different. It should be pointed out that no hard conclusions can be drawn out of the comparison because of three reasons. First of all the architecture of the Spartan-3 as used for the Production Cell setup is very different from the Virtex-5. Second, the designs are different for instance different PID constants and a different number of bits are used. Third, different design

Block	FPGA resource		
	Slices	Flip-flops	LUTs
Processor interface	39	93	53
Error calculation	16	0	26
PID controller	568	96	1291
PWM generator	103	16	208
Encoder interface	62	41	146
Other	351	27	349
Total resource usage	700	312	2073

Table 6.1: FPGA resource consumption of controller design

tools were used namely Handel-C tools for the Production Cell and Xilinx tools for the JIWY. The original Handel-C design can be recompiled for the Virtex-5 but as mentioned in section 2.5 for the floating-point implementation of the Production Cell, recompilation takes some design effort.

An integer implementation for the Production Cell setup required in total 8500 LUTs and 3300 flip-flops for six parallel implementations on a Spartan-3 (Groothuis and Broenink, 2009). The exact distribution of the LUTs and flip-flops over Spartan-3 slices is not known. Translated to Virtex-5 quantities, the integer design used 8500/1.6=5300 LUTs and 3300 flip-flops for six motors. A single motor requires respectively 5300/6=900 LUTs and 3300/6=550 flip-flops. These results are shown in Table 6.2.

Demonstrator	FPGA resource	
	Flip-flops	LUTs
Jiwy	312	2073
Production Cell	550	900

Table 6.2: Jiwy FPGA resource usage compared to integer Production Cell

Compared to the integer implementation of the Production Cell, the JIWY design requires twice the number of LUTs and half the number of flip-flops. In FPGA designs computations are carried out using LUTs. One would expect that the size of a fixed-point implementation is close to the size of an integer implementation. This is because fixed-point is a special form of integer but it requires additional logic for placing the binary point at the correct place for multiplications, addition and subtraction are the same. The numbers reflect that the fixed-point implementation is larger than the integer implementation of the Production Cell. Hard conclusions can not be drawn out of the numbers because the FPGAs and designs are very different. For judging the quality of the code generated by SysGen design, the same design would have to be made in fixed-point using a HDL which has not been done.

7 Conclusions and Recommendations

7.1 Conclusions

The main goal of this assignment is to investigate the use of the ML510 for application in robotics. The ML510 platform is suitable for robotics which is proven by the successful implementation of the JIWY controller design and the implementation of Linux for use on the ML510.

The CE design flow has been extended with FPGA design tools. These design tools work but they have two limitations. First of all the design tools have a steep learning curve and are not user friendly. Second, users need a large amount of computer engineering knowledge, compared to what is needed for 20-sim code generation, before they can use the tools for creating designs for the ML510.

For interfacing the ML510 with CE setups, the board is limited to the use of special connectors. An extension board was acquired which makes use of these connectors, but this board has performance limitations which limit the use of the I/O board to interfacing with setups which have low accuracy requirements.

The third task set for this assignment is the creation of a robotic demonstrator in which the position a motor is controlled by the Virtex-5. The demonstrator works but it does not exactly reach the set point due to limitations in the ML510 I/O board.

The amount of computation power offered by the ML510 is overkill for solving basic control problems. Instead, the platform should be used in applications which can take advantage of the computation power of the ML510.

To summarize, the ML510 is a development platform suitable for use in robotic applications, but at this point the platform is only accessible to designers with a large amount of computer engineering knowledge and to applications with low requirements on I/O performance.

7.2 Recommendations

The ML510 platform is a promising platform for use in robotics but at this point the board is not usable for most people within CE. Below recommendations are given for lowering the boundaries for using the board and recommendations are given for taking better advantage of the platform:

- **Develop a CPU programmable PID controller core**
  In most robotic and mechatronic projects at CE demonstrators are created as a mean for proving a theory. The 20-sim code generator in combination with 4C is used in such projects for rapid prototyping. In projects where hard real-time behavior is important the best option is to implement a controller in configware but this requires a large amount of design effort and computer engineering knowledge. As pointed out earlier in this report, tight coupling of a CPU and FPGA logic allows for designs in which tasks can be performed using a mix of software and configware. A solution for making rapid prototyping possible, while preserving hard real-time behavior, is to design a PID controller core which is configurable by a CPU using a 'control bus' as depicted in Figure 7.1. The figure is based on Figure 5.3 but a CPU and control bus have been added, the text for new parts is in black, gray is used for the original texts. The complete PID controller would be implemented in configware but PID parameters and the sample frequency would be adjustable from

software. A software program would configure the PID parameters at startup.

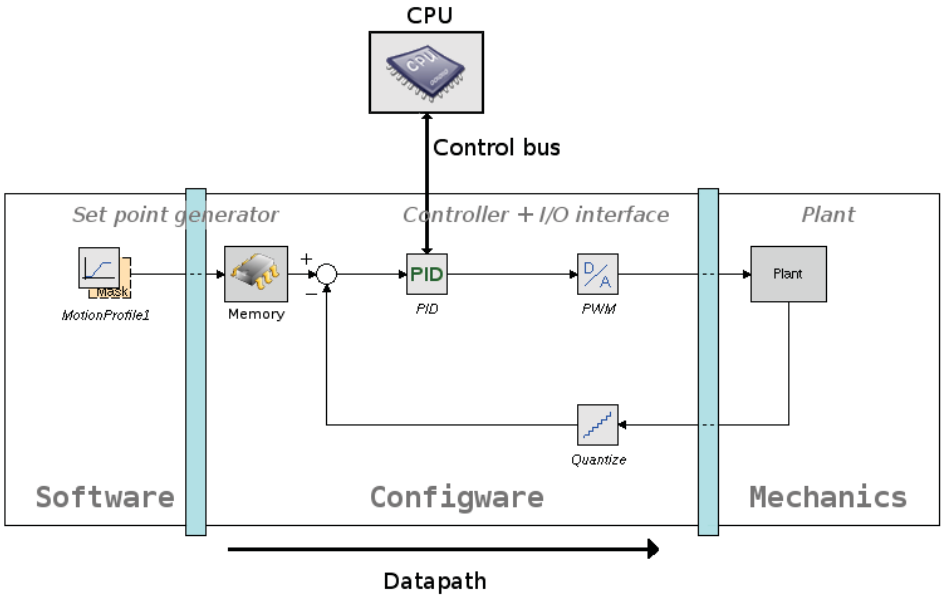

Figure 7.1: Software/configware partitioning containing a programmable PID controller

- **Use the ML510 for complex control tasks**
  The ML510 board offers a large amount of parallel computation power. This computation power should be exploited in complex control tasks like providing vision to robots.

- **Develop new ML510 I/O board**
  The ML510 I/O board, developed at the TU/e, does not perform well for high-frequent PWM signals. This limits accuracy of digital-to-analog conversions. A new I/O board should be developed to fix these performance issues. The board needs to handle level shifting because the voltage level used by ML510 pins is 2.5V. Next, the FPGA pins should be buffered to prevent a test setup from damaging the FPGA. The buffers could be implemented using high-speed optocouplers but if galvanic separation is not needed for instance transistors could be used.

- **Investigate the use of DSP blocks for controller design**
  As described in section 2.5, DSP blocks look promising for the implementation of controllers. It should be investigated whether DSP blocks are indeed as suited for controller design as predicted. A suitable test case would be to port the Production Cell controller design to the ML510 where the DSP blocks can be used for implementation of PID controllers. For taking better advantage of the unused computation power of the ML510, the Production Cell setup could be extended with vision which could be used for sorting the blocks.

- **Investigate the use of ImpulseC as a replacement for Handel-C**
  For implementation of the controllers of the Production Cell setup in an FPGA, Handel-C was used as the HDL. The future of the language and tools is unknown because its developer has been acquired by a competitor which offers a similar

product. A relative new HDL tool which allows for compilation of C to HDL-code is ImpulseC (Impulse Accelerated Technologies, 2009). Its developer works closely together with Xilinx. Due to this cooperation the ImpulseC tools integrate well with Xilinx tools and the tool also supports advanced PowerPC 440 functionality like the APU bus for the creation of custom CPU instructions. ImpulseC might be a good replacement for Handel-C, but it is not known how it handles HandelC constructs like par. The tool could also be used for translating 20-sim C/C++ code to HDL-code.

- **Use 'PLB to Wishbone bridge' for reusing Wishbone designs**
  In another FPGA project carried out within CE (Hettinga, 2008), the configware design made use of the open source communication bus called Wishbone. A large number of free configware designs is available for this bus, which allows for core reuse and thus saves design time. The PowerPC 440 CPU uses PLB busses for communication. A PLB to Wishbone bridge has been developed by the open source community which allows interfacing of Wishbone cores to a PowerPC 440 CPU (OpenCores, 2009). The quality and status of the Wishbone bridge is not known but it might allow CE to standardize on Wishbone cores for PWM and encoder interfaces.

A PowerPC Linux kernel

This appendix describes the ML510 PCI driver which has been written as part of this assignment. The content of this appendix is very technical and is written for people who want an understanding of the PowerPC Linux kernel for debugging or writing a driver. The first section gives an overview of the boot process of the PowerPC Linux kernel, knowledge of this is important for writing drivers. The second section describes the PCI driver which was written as part of this assignment.

A.1 PowerPC Linux boot process

The PowerPC Linux kernel is started in two phases. At system start a 'boot loader' which resides in arch/powerpc/boot is loaded. It performs initial system initialization and decompresses a Linux kernel image to system memory. Once the Linux kernel image is loaded into system memory, the instruction pointer is set to the memory address of the Linux kernel image to start the actual Linux kernel.

The boot loader for the Virtex-5 is contained in arch/powerpc/boot. The files used for the Virtex-5 are described in the order in which they are used at boot:

1 fixed-head.S and crt0.S
  These files contain the initial assembler code for bringing the system up.
2 simpleboot.c
  The code in this file is called by the assembler code to bring up firmware-less systems. Platform specific functions can be called from here to set up caches, to initialize a UART and so on. Further the code also parses the DTS-file.
3 virtex.c
  Contains a Xilinx Virtex-5 specific initialization function for initializing the UART. Information on register offsets is obtained from the DTS-file.
4 main.c
  This file contains the code for decompressing the compressed Linux kernel image. The code outputs debug information to the UART e.g. 'Allocating 0xX bytes for kernel ...' and 'gunzipping (0xX <- 0xY:0xZ)'. After the file is done it moves the instruction pointer to the Linux kernel image and from then on the boot code of the actual Linux kernel is started.

The actual PowerPC Linux kernel code resides in arch/powerpc/kernel. The code makes use of the DTS-file and calls platform specific code in arch/powerpc/platforms/44x. Platform specific code is called using a static structure 'ppc_md' which contains function pointers. During the boot process these functions are set to platform specific functions e.g. functions in arch/powerpc/platforms/44x/virtex.c for the Virtex-5. The generic PowerPC linux code residing in arch/powerpc/kernel calls ppc_md function pointers from different stages of the boot process.

Below some important files used throughout the PowerPC Linux boot process are described:

1 arch/powerpc/head_44x.S
  After the boot loader sets the instruction pointer to the Linux kernel image, the assembler code contained in 'head_4xx.S' is called. It calls code for the initial initizializatioon of interrupts and the memory management unit. It also calls a function 'machine_init' which hooks platform specific and board peripherals. After initial system initialization control is passed to the 'start_kernel' function of Linux kernel init code residing in 'init/main.c'.
2 arch/powerpc/setup_32.c

This file contains the 'machine_init' function which is called from the assembler code. The function parses the device tree and after doing this calls 'probe_machine' for interfacing the ppc_md structure to platform specific functions, which in case for the Virtex-5 are located in arch/powerpc/platforms/44x/virtex.c. After interfacing the ppc_md structure, the probe_machine calls 'ppc_md.probe' which is described below.

3 arch/powerpc/platforms/44x/virtex.c
This file contains Virtex-5 specific functions and information which are hooked up to the ppc_md structure at some point during loading of the Linux kernel. One of the functions in the file is '.probe()' for initializing the Virtex-5 and peripherals which can be loaded into the Virtex-5 like a UART and interrupt controller.

4 init/main.c
The 'start_kernel' is called after bringing up the hardware platform for starting the Linux kernel. The code here initializes interrupts, the memory management unit, the scheduler and other low-level tasks. After completing these low-level tasks the Linux kernel is ready for loading device drivers and once these are loaded, the system is ready for use and hands control over to user space.

A.2 ML510 PCI driver

This section describes the PCI driver which has been written for the ML510 to make PC functionality and PCI slots available.

Most of the PC functionality of the ML510 is provided by the ALI M1533 chipset. The chipset is a PCI device and together with the PCI slots on the ML510 it is connected to the FPGA. The use of PCI requires a PCI soft-core as illustrated in Figure A.1.

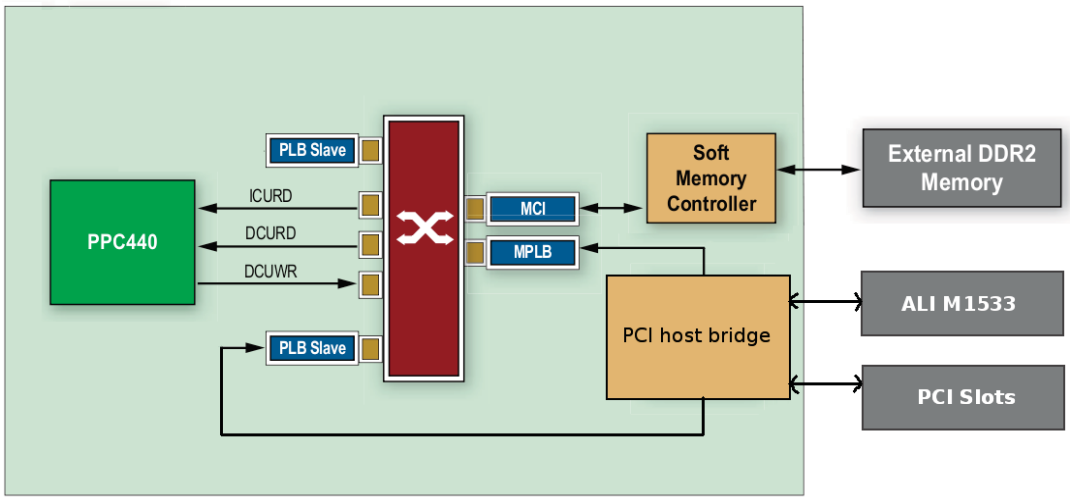

Figure A.1: PCI soft-core connected Crossbar

As can be seen in the figure, the soft-core is connected to the ML1533 chipset and to the PCI slots. Inside the FPGA it is connected as both a master and slave to the PCI bus. The slave connection is required by the CPU for accessing the PCI device by read/write transactions. The master connection is required because PCI device need *Direct Memory Access* (DMA) for bulk read and write operations, so that the CPU does not have to perform all read and write transactions itself. This improves performance because the CPU can do other things while the PCI device is transferring data.

The ML510 PCI driver is responsible for configuring the PCI soft-core and it provides

functionality to the Linux kernel for accessing the PCI bus. The core of the PCI driver is approximately 150 lines of C-code. Next to a PCI driver other changes in the Linux kernel are needed for setting up PCI interrupts and for initializing PCI devices. The complete ML510 PCI code consists of five parts:

- PCI read / write functions
  Generic Linux PCI code is responsible for tasks like scanning the PCI bus and initializing PCI devices. Each PCI driver has to provide functions for reading and writing data to the PCI bus.
- DMA code
  Setting up DMA on the PCI soft-core is complicated. First of all the PCI soft-core needs to be connected properly to the PLB bus in the EDK if this is not done (which is the case in some old Xilinx examples) DMA will not work. Code is required to configure the PCI soft core for DMA, so that PCI devices can access system memory.
- Interrupt mapping in the DTS-file
  PCI uses an interrupt scheme consisting of IRQ lines named A, B, C and D which are wired in a special order to the PCI slots. A table for the interrupt mapping needs to be added to the DTS-file to get PCI interrupts working. On a normal PC motherboard this information is fixed inside the BIOS.
- Interrupt controller code for i8259
  The PCI devices integrated into the ALI M1533 southbridge use a i8259 interrupt controller which is inside the southbridge. This interrupt controller is physically interfaced to the main Xilinx interrupt controller driver. Some special code was required to let the two interrupt controllers cooperate else ALI M1553 peripherals would not work.
- Initialization code for ALI M1533 southbridge
  On a normal PC motherboard the BIOS is responsible for initializing the southbridge chipset at system startup. In case of the ML510 the Linux kernel has to perform this work which includes enabling of PCI devices and assignment of interrupts.

For reference the ML510 PCI support is spread over the following files of the Linux 2.6.31 kernel:

- arch/powerpc/boot/dts/virtex440-ml510.dts
- arch/powerpc/platforms/44x/virtex.c
- arch/powerpc/platforms/44x/virtex_ml510.c
- arch/powerpc/platforms/sysdev/xilinx_pci.c

B Installation of Embryo and Linux on the ML510

Section 5.3 gives an overview of how Embryo and Linux are installed on the ML510. This appendix describes the installation of Embryo and Linux in detail. For reference the figure of Figure B.1 is repeated. The sections in this appendix follow the numbering used in the figure. Next to this appendix some additional information can be found at the Xilinx Linux wiki at `http://xilinx.wikidot.com/`.

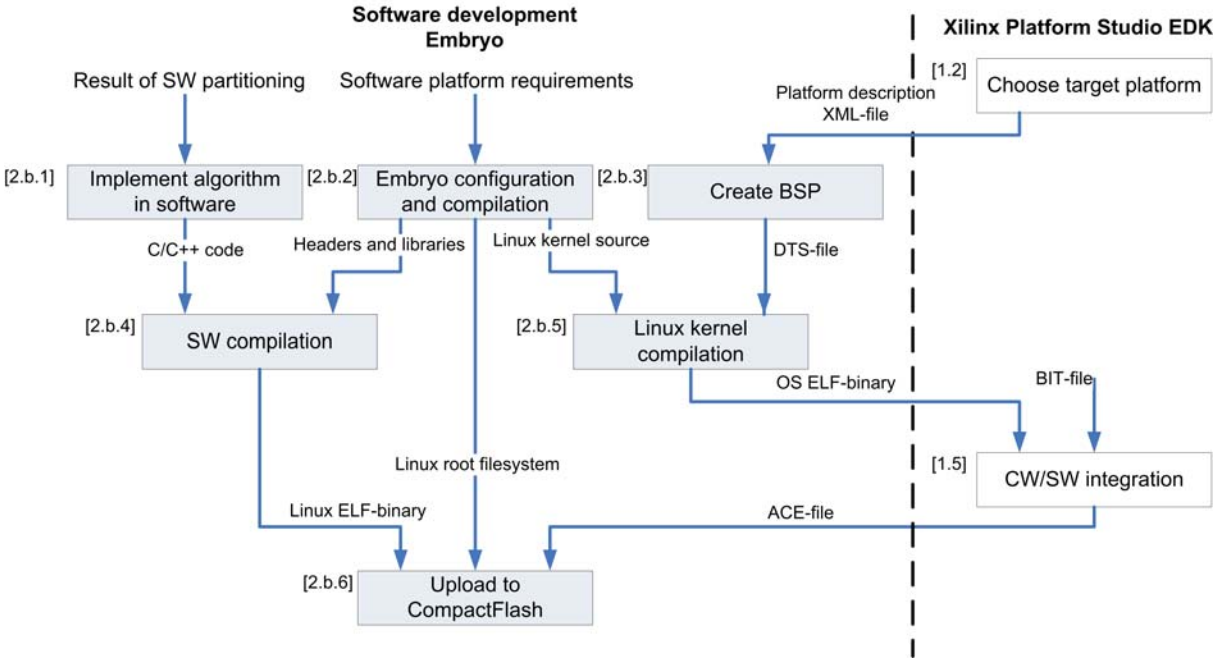

Figure B.1: Embryo flow

B.1 Create Linux compatible BIT-file [1.2 - 1.4]

Xilinx Platform Studio EDK is the recommended tool for creating a system design consisting of a CPU (PowerPC or MicroBlaze) in combination with soft-core periperhals. Once the design is finished a BIT-file can be generated which can be programmed along with software (in the form of ELF-binaries) into the FPGA.

A collection of reference BSB designs and tutorials on how to create these designs from scratch is available from the ML510 website. This appendix describes how to create a basic BSB using EDK 10.1 which is able to run Linux and how to use an existing BSB from Xilinx which has PCI support. Some of the steps might be different in the latest EDK 11.1 which became available at the end of this project.

B.1.1 Basic Linux capable BSB

This guide explains how you can create a ML510 BSB from scratch which can run Linux. The BSB will offer support for a Uart for a serial console, Ethernet and SysACE (for compact flash). Refer to the other design below for features like VGA/DVI and PCI.

Launch the EDK. When the EDK is loaded you are greeted with a dialog which asks whether you want to create a new project or open an existing one. Select 'Base System

Builder Wizard (recommended)' to create a new design. The steps below guide you through the base system builder wizard.

1 Base System Builder - Welcome
   - Select 'I would like to create a new design'.
2 Base System Builder - Select board
   - Board vendor: Xilinx
   - Board name: Virtex-5 ML510 Evaluation Platform
   - Board revision: C
3 Base system builder - Select processor
   - Processors: PowerPC
4 Base system builder - Configure PowerPC processor
   - Processor clock frequency: 300MHz (400MHz should also work but hasn't been tested)
   - Processor bus clock frequency: 100MHz
   - Debug I/F: FPGA/JTAG
   - Cache setup: Enable
   - Enable floating point FPU: Disable (I haven't tried APU floating point in Linux yet)
5 Base system builder - Configure IO interfaces (1 of 4)
   - RS232_Uart_1: Enable o Peripheral: XPS_UART_16550 o Configure as: Uart 16550 o Use interrupt: Enable
   - RS232_Uart_2: Disable
6 Base system builder - Configure IO interfaces (2 of 4)
   - SPI_EEPROM: not needed
   - LEDs_4Bit: not needed but useful for debugging
   - LCD_OPTIONAL: not needed but useful for debugging
   - IIC_EEPROM: not needed
7 Base system builder - Configure IO interfaces (3 of 4)
   - FLASH: not needed
   - Ethernet_MAC: Disable
   - Hard_Ethernet_MAC: Enable o DMA Present: Enable scatter gather DMA (No DMA isn't supported by the Linux driver; enabling it results in LocalLink mismatch errors)
8 Base system builder - Configure IO interfaces (4 of 4)
   - DDR2_SDRAM_DIMM0: Enable o Peripheral: PPC440MC_DDR2
   - SysACE_CompactFLASH: Enable (Not used in the default Linux kernel config but will be used in the future for storing a root file system on compact flash) o Use interrupt: Enable
9 Base system builder - Add internal peripherals (1 of 1)
   - xps_bram_if_cntlr_1 o Memory size: 32kB
10 Base system builder - Cache setup
   - Select the memory peripherals you would like to cache o Enable caching for DDR2_SDRAM_DIMM0 instruction and data cache
11 Base system builder - Software setup
   - Devices to use as standard input, output, and boot memory o STDIN: RS232_Uart_1 o STDOUT: RS232_Uart_1
   - Boot memory: xps_bram_if_cntlr_1 (bram is needed for booting an OS from SysACE)
   - Sample applications: none needed

B.1.2 Xilinx BSB with PCI support

As mentioned in the introduction Xilinx offers various reference BSB designs for the ML510. One of them is called 'ML510 BSB1 Pcores Design' which uses all (or nearly all) periperhals on the ML510 including PCI, VGA/DVI, ethernet and more.

A tutorial on how to create this design and the files can be obtained from the pages below:

- Xilinx BSB1 Pcores Design tutorial at:
  `http://www.xilinx.com/products/boards/ml510/ml510_10.1_3_1/docs/ml510_bsb1_ppc440_pcore_addition.pdf`
- Xilinx BSB1 Pcores Design files at:
  `http://www.xilinx.com/products/boards/ml510/ml510_10.1_3_1/files/ml510_bsb1_pcores_ppc440.zip`

The tutorial (revision February 2009; a previous version was more broken) has one bug which prevents PCI from functioning correctly on Linux. Below a description of the problem.

The PCI bus is implemented by a soft-core which is connected to the PLB bus which is seen by the CPU at 0xa0000000. Both the PLB and PCI have their own address domain. The PCI soft-core performs translation between the two domains. The CPU is able to perform read/write actions to the PCI device but the PCI device is also able to initiate read/write transfers to system memory (DMA).

The Xilinx plbpci doc mentions 'the number of high-order bits substituted in the PLB address presented to the bridge is given by the number of bits that are the SAME between C_IPIFBAR_N and C_IPIF_HIGHADDR_N.'

For the default ML510 pci sample this means:

```
C_IPIFBAR_0          = 0xa0000000
C_IPIF_HIGHADDR_0    = 0xbfffffff
C_IPIFBAR2PCIBAR_0   = 0x00000000
C_IPIFBAR_1          = 0x94000000
C_IPIF_HIGHADDR_1    = 0x97ffffff
C_IPIFBAR2PCIBAR_1   = 0x00000000
```

This means that a CPU write to 0xa0001234 translates to 0x00001234 on the PCI bus and that the pcibar_0 base and pcibar_1 base are zero. As mentioned both the CPU and a PCI device can initiate read/write transactions. In order to have this work properly both CPU and PCI need their own address range else the pci host bridge gets confused. Assuming a 4GB layout it is common to map e.g. the upper 2GB (0x80000000-0xffffffff) for inbound request (CPU reads/writes) and the lower 2GB for outbound requests (reads/writes from a PCI device) and address 0 needs to map to address 0 of the system RAM (for DMA).

The problem is that in the default version of this BSB the inbound and outbound windows overlap. (As indicated a write to 0xa0001234 translates to 0x1234 on the PCI side, this should map to at least 0x80001234 in order not to confuse the PCI host bridge). In order to fix the issue select the PCI soft core in the EDK and modify its settings:

```
set C_IPIFBAR2PCIBAR_0 to 0xa0000000
```

The Linux kernel maintainers wanted me to use 0xa0000000 instead of 0x80000000 to prevent confusion as the base address for pci bar 0 as seen from the CPU is also 0xa0000000.

Note while you can generate a DTS-file for this BSB, I would recommend to use the 'virtex440-ml510.dts' file which I wrote and which should hopefully be part of the 2.6.31

kernel. It contains a hand written pci section containing the memory addresses and interrupt mapping of all PCI peripherals like USB, IDE, Audio and the pci slots.

B.2 Implement algorithm in software and SW compilation [2.b.1, 2.b.4]

The cross-compilers of Embryo can be used for compiling software for use on the ML510. The compile tools can be found in the build directory and can be called from there. They are located in:

```
staging_dir/toolchain-powerpc_gcc-4.1.2_uClibc-0.9.29/usr/bin
```

A better method is to add self written code as a package to Embryo. It would be possible to use standard GNU makefiles and the Embryo build environment takes care of cross-compiling.

The first step is to add create a package directory and edit the needed files. This can be done by copying a directory structure e.g. the one from 'gpio'. The second step is to select the package for compilation. The package appears in 'make menuconfig' under the category defined in the package Makefile. The package is selected for compilation by marking it with a '*'.

The package is compiled and bundled with the Linux root file system using:

```
make V=99
```

If only the package needs to be recompile the following can be used:

```
run make V=99 /packages/[name]/compile'
```

The compiled ELF-binary can then be found in:

```
build_dir/linux-xlnxppc44x_ml510/[package_name].
```

B.3 Embryo configuration and compilation [2.b.2]

This appendix describes how to build Embryo from source and install it on a compact flash card of which the ML510 can boot it.

B.3.1 Embryo Configuration

Obtain Embryo using:

```
svn co https://cewiki.ewi.utwente.nl/svn/EMBRYO/branches/openwrt_ce
```

For the ML510 target, we have a default configuration (set of packages) as starting point. Copy this configuration from the configs/ dir via:

```
grs@ce151:~/embryo$ cp configs/powerpc-xlnx_ml510-2.6 .config
```

Install some necessary dependencies:

```
sudo apt-get install flex gawk zlib1g-dev libncurses5-dev
```

Include additional packages not included by default. You will need this for the 20-sim 4C discoverydaemon:

```
grs@ce151:~/embryo$ make package/symlinks
```

To update the default config for new options/packages in Embryo/OpenWRT:

```
grs@ce151:~/embryo$ make oldconfig
```

You will see for new options questions like:

```
Mount flash in /flash (PC104_MOUNT_FLASH) [Y/n/?] (NEW)
```

In most cases you can just press <enter>. The default value (in this case Y) is almost always ok. If you want to know more about a specific option press ? for help.
To review and/or change the settings for your target, you can always run:

```
grs@ce151:~/embryo$ make menuconfig
```

To see which version of Embryo was used to compile the image running on a particular ML510 board:

1 Start embryo on the ML510
2 Login with root and password embedded
3 Run:

```
cat /etc/embryo/revision
```

B.3.2 Linux kernel configuration

By default Embryo compiles a Linux kernel with support for a limited number of ML510 peripherals. If more peripherals and additional drivers are needed, the Linux kernel configuration can be adjusted. The Linux kernel can be configured from within the Embryo environment after Embryo is configured using:

```
make kernel_menuconfig
```

A list of drivers required for some periperhals is shown in Table B.1.

Peripheral	Driver
Audio	ali5451
Ethernet	ll_temac
IDE	ali15x3
USB	ohci-hcd
SysAce	xsysace
Video	xilinxfb

Table B.1: Linux drivers required for different peripherals

All of the drivers mentioned in the table have been tested except for the audio driver. In theory this driver should work but if it does not some initialization code could be incorrect in 'arch/powerpc/platforms/44x/virtex_pci.c'. Further for USB the ohci-hcd driver should be set to little endian mode and the big endian options should be disabled because all PCI on the ML510 is little endian.

B.3.3 Embryo Compilation

After Embryo Configuration, the next step is to compile everything via:

```
grs@ce151:~/embryo$ make V=99
```

This will take a while, so time for coffee...

When the compilation process finishes without errors, you can find the PC/104 embryo output files in bin/.

```
xlnxppc44x-rXXXX-config (Embryo .config file)
xlnxppc44x-rXXXX-rootfs.ex2 (16 MB EXT2 filesystem image)
xlnxppc44x-rXXXX-rootfs.tgz (inhoud rootfs als tarball)
xlnxppc44x-rXXXX-zImage (PowerPC zipped kernel image)
simpleImage.virtex440-ml510.elf (PowerPC ELF kernel image)
```

You will need these files for ACE-file generation as described in section B.5 and installation as described in section B.6.

B.4 Create BSP [2.b.3]

The PowerPC Linux kernel uses a mechanism consisting of 'device tree' files (.dts) which contains all info (e.g. memory addresses, interrupts) of board peripherals. At the end of Linux kernel compilation the dts file is combined with the compiled kernel sources to construct a Linux kernel. The contents of the dts file depends on the used soft-cores and can be generated from XPS using a 'device-tree' plugin which is available from http://git.xilinx.com.

Obtain the device-tree files using:

```
git clone git://git.xilinx.com/device-tree.git
```

Once the files are downloaded it should be possible to add the device-tree generator to your current project using:

```
cp -r bsp <path to project>/
```

On my own system this didn't work and I added it globally to /opt/Xilinx/10.1/EDK/sw/lib/bsp by copying the contents of the bsp directory to here. The result is the following directory tree:

```
/opt/Xilinx/10.1/EDK/sw/lib/bsp/device-tree/data/device-tree_v2_1_0:
    device-tree_v2_1_0.mld
    device-tree_v2_1_0.tcl
```

Once device-tree is installed properly it should appear in 'Software âĘŠ Software Platform Settings' in Xilinx Platform Studio EDK. Select the 'device-tree' option in this menu.

If no console device is selected on the 'OS and Libraries page' make sure to set 'console device' to 'RS232_Uart_1' (in case of the design made above) else not all boot output appears on the RS232 port. (To be exact this option sets linux,stdout-path in the dts file)

In order to create a .dts file 'Software → Generate Libraries and BSPs' in the EDK (as of version 11 of the Xilinx software, Platform Studio SDK should be used for this task). A dts file should now appear in 'ppc440_0/libsrc/device-tree' called xilinx.dts. This file should be put in the Linux kernel source in arch/powerpc/boot/dts and it should be naded: 'virtex440-ml510.dts'.

When SysACE is enabled in the FPGA config and you want to boot the root file system from a partition on the compact flash card add 'root=/dev/xsa2' for partition 2 on the compact flash card. For booting busybox you also need a line which adds 'init=/etc/preinit'.

B.5 Build Embryo ACE file - CW/SW integration [1.5]

This section describes how to create an ACE-file which is used for booting the ML510 board. This file contains a hardware mapping in the form of a BIT file and a software mapping consisting of a ELF binary. The resulting ACE file can for instance be put on a compact flash card in order to boot the board.

An ACE-file is created using the Xilinx Memory Debugger (XMD) in combination with TCL script called 'genace.tcl'. When using the script you need to pass a so called '.opt' file which contains the name of the BIT and ELF files to use among other options like the type of CPU to use.

The script is used like this:

```
xmd -tcl genace.tcl -opt gen.opt
```

A sample .opt file is shown below which works for a single processor for the ML510:

```
-jprog
-target ppc_hw
-hw ./download.bit
-elf ./simpleImage.virtex440-ml510.elf
-board ml507
-ace boot.ace
```

Note that the board type is set to ml507 because older scripts don't recognize the ML510 (at least this is the case with scripts included in EDK 10.1 and 11.1) but this is no problem as the ML507 contains a similar FPGA with a lower number of logic elements and 1 CPU instead of 2. The Linux kernel image can be found in 'arch/powerpc/boot' of the Linux kernel source and is called 'simpleImage.virtex440-ml510.elf' (e.g openwrt_ce/build_dir/linux-xlnxppc44x_ml510/linux-2.6.30-rc7/arch/powerpc/boot/simpleImage.virtex440-ml510.elf) when using the 'virtex440-ml510.dts' file for the ML510 reference design. The file download.bit is the BIT-file generated using the EDK.

The .opt file for two processors

```
-jprog
-target ppc_hw
-board ml510
-hw ./ml510_bsb_system.bit
-debugdevice cpunr 1
-elf ./simpleImage.virtex440-ml510.elf
-target ppc_hw
-debugdevice cpunr 2
-elf ./Standalone.elf
-ace boot.ace
```

Execute this command from a EDK Shell (XPS → Project → Launch EDK Shell) because a normal shell even when the Xilinx settings scripts are sourced don't have tools like 'powerpc-eabi-objdump' in their path which is needed by the genace script. After running the command a file 'boot.ace' should be created which can be placed on a CompactFlash card.

B.6 Upload to CompactFlash [2.b.6]

Uploading of software and configware to a CompactFlash card requires five steps:

- Partition a CompactFlash card with a FAT partition for the ACE -file and a partition with a Linux file system.
- Create an ACE file which contains the FPGA BIT file and the OS ELF-binary containing the Linux kernel.
- Install the Linux ELF-binary containing the software program e.g. the controller software.
- Install the Linux root file system on the CompactFlash card.
- Install the ACE-file to the FAT partitition of the CompactFlash card.

B.6.1 Partition a CompactFlash card

The Virtex-5 FPGA can be configured using compact flash (SysACE), JTAG and flash memory. This guide assumes that the FPGA is configured from compact flash. Two partitions are required on the CompactFlash card a FAT16 one for configuring the FPGA and a Linux one (e.g. ext3) for storing the Embryo root file system.

When booting the ML510 using SysACE (which happens when SW3=00010101) the SysACE interface looks for a .ace file on a FAT16 partition of the compact flash card. The .ace file contains the FPGA configuration (a .bit file) and software (a .elf binary for the PowerPC or a binary for a MicroBlaze). The Xilinx Linux SysACE block device driver needs to be build into the Linux kernel in order to be able to access the compact flash card. This kernel option can be found under Device Drivers → Block Devices → Xilinx SysACE support.

Start Linux fdisk e.g. fdisk /dev/sdb if the cf-card is /dev/sdb

1 Remove all current partitions (use the d option on each partition)
2 Add a 100MB fat16 partition for storing the .ace file
    1 Add a new partition using the 'n' option, choose number 1
    2 The start cylinder is 1
    3 The end cylinder +100M
    4 Change the file system id to fat16 using the 't' option and pass it value '6' (fat16)
5 Add a linux partition to fill the rest of the cf-card

    Add a new partion using the 'n' option, assign it partition number 2
6 Write back the partition table use 'w'
7 Quit fdisk use 'q'

Now remove the compact flash card and reinsert in order for Linux to see the new partition table. Create partitions on the device

```
mkdosfs -v -F 16 -S 512 /dev/sdb1
```

Reconnect the CompactFlash card first because Linux doesn't have a /dev/sdb2 yet, then:

```
mkfs.ext3 /dev/sdb2
```

The CompactFlash card is ready to use now.

For reference the partition table should look like:

```
Disk /dev/sdb: 512 MB, 512483328 bytes
16 heads, 63 sectors/track, 993 cylinders
Units = cylinders of 1008 \item 512 = 516096 bytes
Disk identifier: 0x37191e13
  Device Boot      Start         End      Blocks   Id  System
/dev/sdb1               1         204      102784+   6  FAT16
```

```
/dev/sdb2          205          993        397656    83   Linux
```

B.6.2 Install the Linux ELF-binary

Copy the Linux ELF-binary to a directory of choice on the second partition containing the Linux filesystem.

B.6.3 Install the Linux root filesystem

Copy the contents from the xlnxppc44x-rXXXX-rootfs.tgz to the second partition of the Compact Flash card.

B.6.4 Install the ACE-file

Copy the ACE-file to the FAT partition of the CompactFlash card. If there is only one ACE file on this partition it will be booted by default.

C SysGen implementation of PID algorithm

This appendix shows an implementation of the PID algorithm in SysGen. It implements the PID algorithm as used by 20-sim:

```
factor = 1 / ( sampletime + tauD * beta );
uD = factor * ( tauD * previous ( uD ) * beta
     + tauD * kp * ( error - previous ( error ))
     + sampletime * kp * error );
uI = previous( uI ) +  sampletime * uD / tauI;
output = uI + uD;
```

The SysGen implementation of the PID algorithm is shown in Figure C.1. It uses the paramters: Kp=10, tauD=0.02 s, beta=0.1, tauI= 10 s and Ts=1 ms.

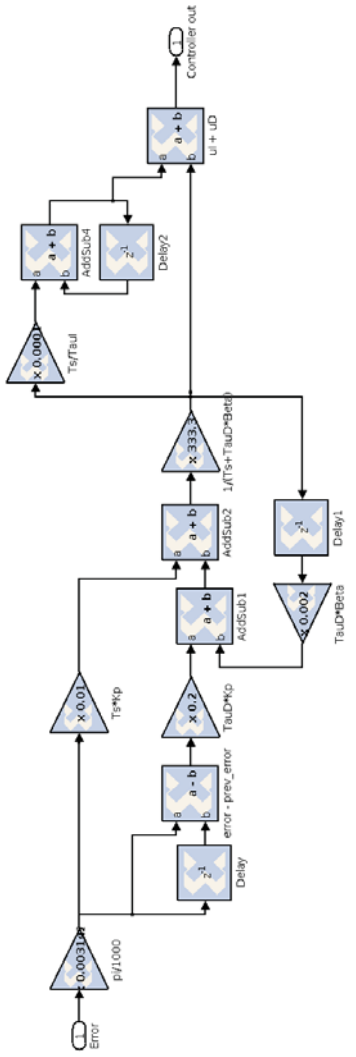

Figure C.1: PID-controller implemented using Xilinx blocks im Simulink

Bibliography

Abramson, C., D. Isaacs and A. Ansari (2008), Embedded Processing Innovations with Virtex-5 FXT Devices, in *Xcell Journal Issue 64: Embedded Processing Solutions: Inside the New VirtexÂő-5 FXT FPGA*, Xilinx, pp. 8–13.
http://www.xilinx.com/publications/xcellonline/xcell_64/xc_pdf/p08-13_64-cover.pdf

Agility Design Solutions (2009), Mentor Graphics Aquires Agility C Synthesis Suite.
http://www.agilityds.com/literature/012209_MentorAgilityReleasevf2.pdf

Beckhoff (2008), Ethercat IP for Xilinx FPGAs datasheet.
http://www.beckhoff.com/download/Document/EtherCAT/Development_products/EtherCAT_IPCore_Xilinx_Datasheet_all.pdf

Broenink, J. F., M. A. Groothuis, P. M. Visser and B. Orlic (2007), A Model-Driven Approach to Embedded Control System Implementation, in *Western Multiconference on Computer Simulation WMC 2007*, Eds. E. J. Anderson and R. Huntsinger, San Diego, USA, pp. 137–144, ISBN 1-56555-311-X.
http://www.ce.utwente.nl/rtweb/publications/2007/pdf-files/102CE2007_WMC07_pdf.pdf

Colenbrander, R., A. Damstra, C. Korevaar, C. Verhaar and A. Molderink (2008), Co-design and Implementation of the H.264/AVC Motion Estimation Algorithm Using Co-simulation, in *Proceedings of the 11th Euromicro Conference on Digital System Design*, IEEE Computer Society Press, Los Alamitos, pp. 210–215.
http://doc.utwente.nl/64995/

Controllab Products (2009), 20-sim.
http://www.20-sim.com/

Corporaal, H. (2006), Embedded system design, in *PROGRESS White Papers 2006*, Technologiestichting STW, pp. 7–27.

Damstra, A. S. (2008), Virtual prototyping through co-simulation in hardware/software and mechatronics co-design, MSc. Thesis 005CE2008, University of Utwente.

Deen, B. (2008), A software solution for absolute position estimation using WLAN for robotics, MSc. Thesis 022CE2008, University of Utwente.

DIAPM (2009), RealTime Application Interface for Linux.
https://www.rtai.org/

Eclipse Foundation (2009), Eclipse.
http://www.eclipse.org

Groothuis, M. A. (2004), Distributed HIL simulation for BodeRC, MSc. Thesis 020CE2004, University of Utwente.

Groothuis, M. A. and J. F. Broenink (2009), HW/SW Design Space Exploration on the Production Cell Setup, in *To be published in Communicating Process Architectures 2009*, IOS Press, Netherlands.

Groothuis, M. A., J. F. Broenink and J. P. van Zuijlen (2008), FPGA based Control of a Production Cell System, in *Communicating Process Architectures 2008*, volume 66, IOS Press, Netherlands, volume 66, pp. 135–148.
http://www.ce.utwente.nl/rtweb/publications/2008/pdf-files/p135-groothuis.pdf

Hartenstein, R. (2006), Morphware and Configware, in *Handbook of Nature-Inspired and Innovative Computing*, Springer US, pp. 343–386, ISBN 978-0-387-40532-2.

Hettinga, S. A. (2008), FPGA configuration development automation, Individual Assignment-Report 031CE2008, University of Utwente.

IBM (2003), Book E: Enhanced PowerPC Architecture.
http://www.freescale.com/files/32bit/doc/user_guide/BOOK_EUM.pdf?WT_TYPE=Users%20Guides&WT_VENDOR=FREESCALE&WT_FILE_FORMAT=pdf&WT_ASSET=Documentation

IBM (2009), Power Architecture.
http://www-03.ibm.com/technology/power/

Impulse Accelerated Technologies (2009), Impulse-C.
http://www.impulseaccelerated.com/

Michalek, D., C. Gehsat, R. Trapp and T. Bertram (2005), Hardware-in-the-Loop-Simulation of a Vehicle Climate Controller with a combined HVAC and Passenger Compartment Model, in *Proceedings of the 2005 IEEE/ASME*, pp. 1065–1070, ISBN 0-7803-9047-4.

Oosterom, H. P. (2006), On the verification of real-time distributed embedded control systems, MSc. Thesis 024CE2006, University of Utwente.

OpenCores (2009), PLBv46 to Wishbone Bridge.
http://www.opencores.org/project,plbv46_to_wb_bridge,overview

OpenWrt (2009), OpenWrt.
http://www.openwrt.org/

Przybus, B. (2007), Leveraging the Virtex-5 SXT High-Performance DSP Solution, in *Solutions for High-Performance Signal Processing Designs Issue 3 : Xilinx Unleashes New XtremeDSP Portfolio*, Xilinx, pp. 6–7.
http://www.xilinx.com/publications/magazines/dsp_03/xc_pdf/p06-07-3dsp-przybus.pdf

QNX Software Systems (2009), QNX Neutrino RTOS.
http://www.qnx.com/products/neutrino_rtos/

Sassen, T. (2009), Floating-point based control of the Production Cell using an FPGA with Handel-C, MSc. Thesis 009CE2009, University of Utwente.

The Mathworks (2009), Simulink HDL Coder.
http://www.mathworks.com/products/slhdlcoder/

Torvalds, L. (2009), Linux.
http://www.kernel.org/

Visser, P. M., M. A. Groothuis and J. F. Broenink (2004), Multi-Disciplinary Design Support using Hardware-in-the-Loop Simulation, in *5TH PROGRESS Symposium on Embedded Systems*, STW Technology Foundation, pp. 206–213.
http://doc.utwente.nl/56327/

Wind River Systems (2009), Wind River VxWorks: Embedded RTOS with support for POSIX and SMP.
http://www.windriver.com/products/vxworks/

Xenomai Project (2009), Xenomai: Real-Time Framework for Linux.
http://www.xenomai.org/index.php/Main_Page

Xilinx (2007), WP284 - Advantages of the Virtex-5 FPGA 6-Input LUT Architecture.
http://www.xilinx.com/support/documentation/white_papers/wp284.pdf

Xilinx (2008a), EDK Introduction.

Xilinx (2008b), Xilinx Announces Development Platform for Building Dual Processor Embedded Systems Using Virtex-5 FXT FPGA.

http://press.xilinx.com/phoenix.zhtml?c=212763&p=irol-newsArticle&ID=1218920&highlight=

Xilinx (2008c), Xilinx ML510.
http://www.xilinx.com/products/devkits/HW-V5-ML510-G.htm

Xilinx (2009a), "Xilinx DS335 Floating-Point Operator 5.0, data sheet".
http://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf

Xilinx (2009b), Xilinx Linux Device Tree Generator.
http://xilinx.wikidot.com/device-tree-generator