University of Twente

EEMCS / Electrical Engineering
Control Engineering

Floating-point based control of the Production Cell using an FPGA with Handel-C

Thijs Sassen

MSc report

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

June 2009

Report nr. 009CE2009
Control Engineering
EE-Math-CS
University of Twente
P.O.Box 217
7500 AE Enschede
The Netherlands

Summary

The current design flow for embedded control applications within the control engineering group is based on models from which code can be generated for different hardware platforms. Tools that are used in this flow are 20-sim, 20-sim 4C and gCSP.

These tools support code generation for CPU based systems; an alternative to the CPU architecture is an FPGA architecture. In a previous project, the Production Cell system was ported from a CPU architecture toward an FPGA architecture with the use of the hardware description language Handel-C. The Production Cell consists of 6 parallel controlled motors which pass around metal blocks. The motors need to synchronize in order to operate correctly. The ported FPGA architecture is based on integer loop controllers due to FPGA-resource restrictions at that time.

The ported system had a major drawback, the loop controllers had to be redesigned for the integer data type in stead of leaving it in floating-point resolution. This costed quite a lot of design effort.

This project describes the design and realization of the Production Cell system using alternative loop controllers for an FPGA architecture. The goal is to find a loop controller which fits in the design flow with minimum FPGA-resource usage and design effort.

The discussed alternative loop controllers are floating-point, fixed-point, hard-core and soft-core with a floating-point unit. The floating-point loop controller is the best one to use, taking in mind the requirements of the loop controller. To minimize the FPGA-resource usage the floating-point loop controller is implemented by controlling the 6 motors in the Production Cell sequentially using one controller.

With this sequential control, the existing Production Cell software framework needs to be redesigned due to the fact that it was based on parallel control. The new Production Cell framework includes an improved safety layer, which can detect deadlock and system errors. This is possible because all sensor and motor information is available in a global safety layer.

The outcome of this project showed that it is possible to control the Production Cell with a loop controller that fits in the current design flow while gaining more precision without using too much FPGA-resources.

During the investigation on alternative loop controllers the soft-core and hard-core alternatives showed to be promising, but could not be investigated, because the FPGA that needs to be used for these alternatives was not operational. Therefore, it is recommended that these alternatives need further investigation as soon as this FPGA is operational.

Samenvatting

Het huidige ontwikkeltraject voor regelapplicaties, binnen de Control Engineering group, is gebaseerd op het maken van modellen voor verschillende hardware platformen waar vanuit broncode gegenereerd kan worden. Programma's die hiervoor gebruikt worden zijn 20-sim, 20-sim 4C en gCSP.

Deze programma's ondersteunen broncode generatie voor CPU gebaseerde systemen. Een alternatief voor een CPU architectuur is een FPGA architectuur. In een voorgaand project is de productiecel omgezet van een CPU gebaseerde architectuur naar een FPGA gebaseerde architectuur door gebruik te maken van de hardware beschrijvingstaal Handel-C. De Productiecel bestaat uit 6 parallel geregelde motors welke blokjes doorgeven. De motoren moeten hun bewegingen synchroniseren om correct te werken. De standaard regelaar van de productiecel is gebaseerd op het floating-point datatype. Echter door FPGA beperkingen tijdens dat project is de regelaar voor de geporteerde FPGA architectuur gebaseerd op het interger datatype.

Het FPGA systeem heeft hierdoor een groot nadeel. Doordat de orginele regelaar gebaseerd is op floating-point moest de regelaar opnieuw ontworpen worden voor het integer datatype. Deze aanpassing heeft tamelijk veel ontwerp tijd gekost.

Dit project beschrijft het ontwerp en de realisatie van de productiecel door gebruik te maken van alternatieve regelaars en tevens geschikt zijn voor een FPGA architectuur. Het doel is om een regelaar te vinden die in het huidige ontwikkel traject past en tevens zo min mogelijk FPGA ruimte en ontwerp tijd kost.

De alternatieve regelaars die besproken zijn tijdens dit project zijn: floating-point, fixed-point, hard- en soft-core CPUs met een floating-point unit. De floating-point regelaar is als beste alternatief uit het onderzoek gekomen. Om de FPGA ruimte zo klein mogelijk te houden is de floating-point regelaar zo gebouwd dat deze 6 motoren sequentieel aanstuurt.

Het huidige productiecel raamwerk is gebaseerd op een parallelle aansturing van de motoren. Omdat de floating-point regelaar sequentieel aanstuurt is het raamwerk aangepast. Het nieuwe raamwerk bevat naast de vernieuwde aansturing ook een veiligheids laag die fouten en wederzijds beïnvloeden van processen detecteert.

Het resutaat van dit project laat zien, dat het mogelijk is de productiecel aan te sturen met een regelaar voor een FPGA architectuur, die in het ontwikkeltraject past en tevens met zo min mogelijk ontwerp tijd en niet al te veel FPGA ruimte gebruikt.

Tijdens het onderzoek naar alternatieve regelaars is gebleken dat de hard- en soft-core alternatieven veel belovend zijn. Omdat er tijdens dit project nog geen FPGA operationeel was waarop deze alternatieven getest konden worden is het aanbevolen om deze alternatieven verder te onderzoeken zodra de FPGA operationeel is waarom deze getest kunnen worden.

Preface

With this report I conclude my study at the University of Twente.

I would like to thank the following people for their help during my study:

Jan Broenink, Marcel Groothuis and Bert Molenkamp for the support and guidance during this project.

My fellow MSc students for explaining basic control engineering aspects and discussing my own theories to bring this project to a higher standard.

My parents, Ado and Femmy Sassen, the rest of my family and girlfriend, Marloes van Schaik, for their confidence and support during this project and my study period.

Thijs Sassen
Enschede, June 2009

Contents

1 Introduction

1.1 Context

The current design flow for embedded control applications within the control engineering group is based on models from which code can be generated for different hardware platforms. Tools that are used in this flow are 20-sim and 20-sim 4C from Controllab Products (2009) and gCSP (Jovanovic et al., 2004). 20-sim is used to make a graphical model representation of a system. With this model the behavior of the system can be simulated and analyzed. With gCSP the communication between parts of the (20-sim) software system can be modeled, visualized and animated.

The current design flow is oriented toward CPU based systems (PC) with an Von Neumann architecture (Silberschatz et al., 2005). An alternative to the Von Neumann architecture is an FPGA (Field Programmable Gate Array). FPGAs are programmable devices that can be used to implement functionality that is normally implemented in dedicated electronic hardware, but can also be used to execute tasks that run normally on CPU-based systems. With an FPGA, true parallel execution, which can not be done with an Von Neumann architecture, high speed and hard real-time behavior can be achieved. A feasibility study of van Zuijlen (2007) showed that the Handel-C (Celoxica, 2009) language is suitable for embedded control applications for creating code for an FPGA architecture. Figure 1.1 shows the design flow for the two architectures.

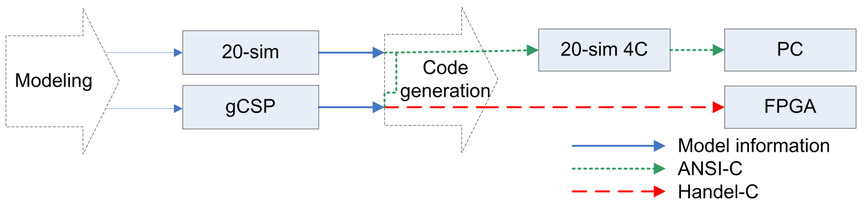

Figure 1.1: Design flow for embedded control applications adapted from van Zuijlen (2008)

This project uses the Production Cell system made by van den Berg (2006) as demonstrator. The 20-sim and gCSP models from Maljaars (2006) and the FPGA framework of van Zuijlen (2008) are used. All these projects are part of the ViewCorrect PhD project of Groothuis (2009).

The 20-sim models made by Maljaars (2006) contain the design for the loop controller, which consist of a motion profile generator and a PID algorithm, for the Production Cell. This design is based on the floating-point data type. Van Zuijlen (2008) ported the Production Cell to an FPGA architecture. This porting is based on integer loop controllers. This implementation involved major rewriting of the existing control algorithm made in 20-sim. The result of this integer-based loop controller is that the design effort drastically increases in time.

1.2 Aim of the Project

The aim of this project is to find a loop controller on an FPGA architecture that will fit in the current model based design flow without making time consuming adjustments to existing loop controller models. Several alternatives are available: floating-point, fixed-point, hard-core and soft-core with a floating-point unit. These alternatives will be investigated and with the best solution regarding to the model based design, the production cell will be implemented.

Since an FPGA has limited FPGA-resources, the challenge lies within making a proper trade-off between timing, FPGA-resource usage, accuracy and design effort.

1.3 Approach

Figure 1.2 shows the design methodology for the Production Cell. Step [1] and [2] are done by van den Berg (2006) and Maljaars (2006). Step [3a], [3b] and [4] are done in this project.

This project is divided into three development stages:
- Investigation
- Implementation and Testing
- Results

Step [3a] contains all development stages and step [3b] and [4] together contain again all development stages.

Step [3a] begins with an investigation on alternative loop controllers. Then the found alternative loop controllers will be implemented and tested. Finally the results of these implementations will then be compared and a loop controller will be chosen to implement the Production Cell with.

Step [3b] and [4] begins with an investigation on which implementation structure to use for the Production Cell framework. After this investigation the Production Cell framework will be integrated with the chosen loop controller implementation. At the end, the results of the new Production Cell System will be elaborated.

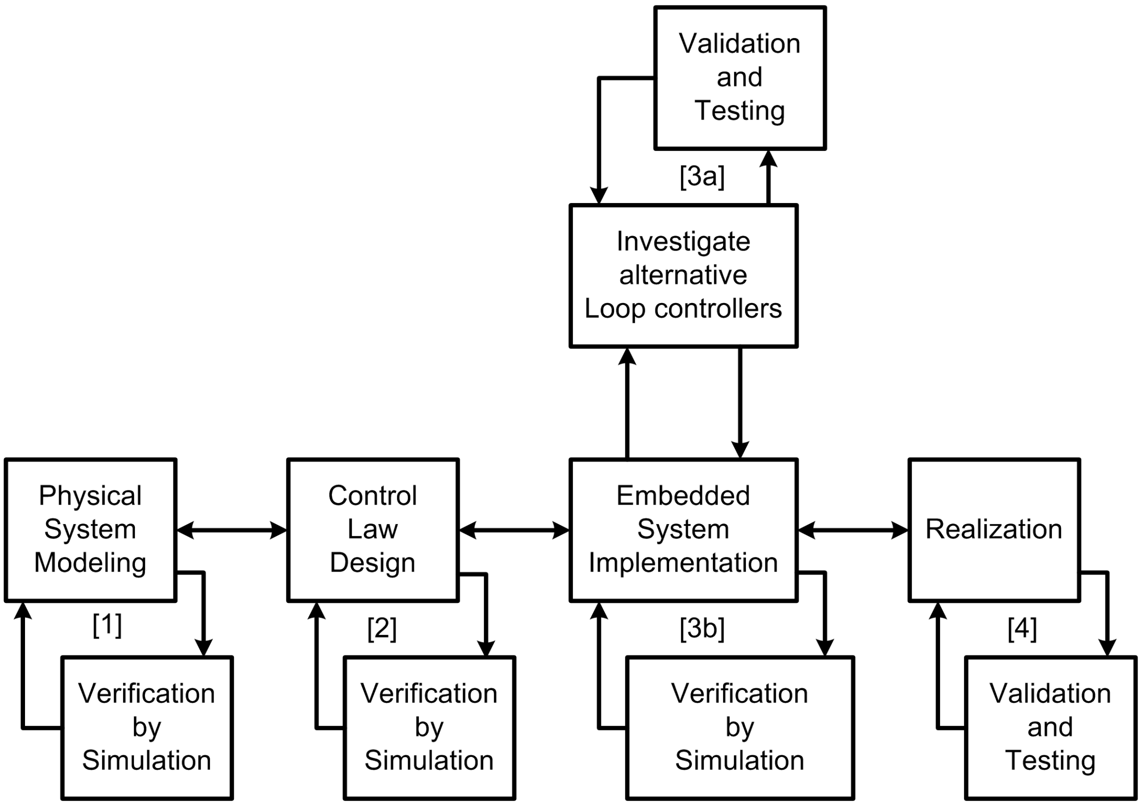
Validation and Testing
[3a]
Investigate alternative Loop controllers
Physical System Modeling
Control Law Design
Embedded System Implementation
Realization
[1]
[2]
[3b]
[4]
Verification by Simulation
Verification by Simulation
Verification by Simulation
Validation and Testing

Figure 1.2: CE design methodology adapted from Broenink and Hilderink (2001)

1.3.1 Research

Groothuis et al. (2008) proposed several alternative loop controllers. These alternatives are listed in table 1.1.

	Alternative	Benefit	Drawback
[1]	Floating point library	High precision; re-use existing controller	Very high logic utilization because each calculation gets its own hardware
[2]	Fixed point library	Acceptable precision	High logic utilization because each calculation gets its own hardware
[3]	Integer	Native data type	Low precision in small ranges; adaptation of the controllers needed
[4]	Soft-core CPU+FPU	High precision; re-use existing controller	High logic utilization unless stripped
[5]	Hard-core CPU+FPU	High precision; re-use existing controller	High price, large footprint of the hard-core CPU leaving less FPGA-resources

Table 1.1: Alternative loop controllers for the Production Cell

The alternatives stated in table 1.1 give a summary of loop controller implementation possibilities. Each alternative will be worked out in a work flow diagram. This way a overview on how the loop controller is implemented can be obtained. All these alternatives will be elaborated in chapter 3.

1.3.2 Implementation and Testing

In order to implement the floating-point and fixed-point loop controllers, arithmetic operations add, multiply, subtract and divide need to be made. Each arithmetic operation will be tested against a similar IEEE 754 floating-point ANSI C (Kernighan and Ritchie, 1988) CPU based implementation. When all operations proved to be working correctly they will be integrated into a math library. After this math library is finished, the motion profile generator and PID algorithm can be implemented. The motion profile generator and PID algorithm will be tested against the 20-sim simulation results of the Production Cell 20-sim loop controller model. The final step is to integrate the loop controller with the Production Cell framework and test the behavior on the real setup.

1.3.3 Results

The results, step [3a] from figure 1.2, are evaluated according to the following requirements: timing, design effort, FPGA resource usage and accuracy.

Timing is an important factor in real-time systems. The Production Cell works with a 1 ms sample time. Groothuis et al. (2008) concluded that there is plenty of time to do calculations on an FPGA within this sample time, hence, all loop controller implementations are expected to meet this requirement.

Design effort will be measured in the amount of time the implementation requires to integrate into the design flow (e.g. the implementation of van Zuijlen (2008) required rewriting of the existing 20-sim loop controller models which was very time consuming, namely ± 4 weeks).

FPGA resource usage will be measured in the number of LUTs (see section 2.1) that are used for the loop controller implementation. Large FPGAs are expensive which means that it is desired to use as little FPGA-resources as possible.

Accuracy will be measured according to maximum loop controller error which must stay below 0.5 mm (Maljaars, 2006). Van Zuijlen (2008) proved that an integer loop controller meets this accuracy requirement and the other suggested loop controllers (table 1.1) are all higher in mathematical accuracy which should imply that these alternatives can meet the requirements also.

The result of step [4] from figure 1.2 is compared to the final result of van Zuijlen (2008) in terms of design effort and FPGA resource usage.

1.4 Used Hardware

The FPGA used in this project is the Xilinx Spartan 3 XC3S1500L-4-FG320. This FPGA contains 26624 LUTs, 589k bit memory and has 32 dedicated 18x18 bits multipliers. For test purposes an FPGA test board (RC10) from Celoxica is used. This board has the same FPGA as used in the Production Cell together with a number of I/O ports to send and retrieve data from the FPGA. Handel-C has support libraries for the I/O interfaces on the RC10 which makes the RC10 feasible for debugging and testing code for an FPGA realy fast. More information about FPGAs and the corresponding abbreviations is explained in section 2.1.

1.5 Report Structure

Background information on this project is provided in chapter two. All the alternative loop controllers is discussed in chapter three. It shows the up and downsides of each implementation. Chapter four contains the redesigned production cell controller with information on its structure and newly added safety layer. The last chapter is chapter five which lists the conclusions and recommendations that are drawn from this project.

Appendix A shows information on a tool that was made to convert the Handel-C floating-point format to the ANSI-C float format. Appendix B shows a table of different Production Cell demos and their capabilities. This appendix also includes a short manual on how to start and stop the Production Cell demo. Appendix C shows information on how to build Handel-C projects. Finally, appendix D shows the location of the source code of the Floating-point based Production Cell system.

2 Background

This chapter contains information about the background of this project. The used platform, implementation language, demonstrator and mathematical representation of numbers in an computer are elaborated. After reading this chapter, people who are unfamiliar with the topics used in this project should be able to understand the rest of the report. The Production Cell section, part of the soft-cores without the Microblaze and hard-cores without the PowerPC part, from the FPGA section, are written by van Zuijlen (2008).

2.1 FPGA

A Field-Programmable Gate Array (FPGA) is a semiconductor device that can be configured by the customer or designer after manufacturing hence the name *field-programmable*. FPGAs are programmed using a logic circuit diagram or source code in a Hardware Description Language (HDL) to specify how the chip should work. They can be used to implement any logical function that an Application-Specific Integrated Circuit (ASIC) could perform, however, the ability to update the functionality after shipping offers advantages for many applications.

Xilinx FPGAs (Figure 2.1) contain Configurable Logic Blocks (CLB), and a hierarchy of reconfigurable interconnects that allow the blocks to be wired together. CLBs can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. Each CLB consists at least one Logic Cell (Figure 2.2) which is defined as one 4-input Look-Up Table (LUT), a flip-flop and a connection to a global clock. FPGAs can also have dedicated memory (Blockram) and multiplier blocks (18x18 bit). These blocks can be used to perform specific tasks to safe CLBs. The FPGA-resource usage is measured in number of used LUTs. In this project an Xilinx FPGA is used, other manufacturers provide FPGAs with a similar architecture.

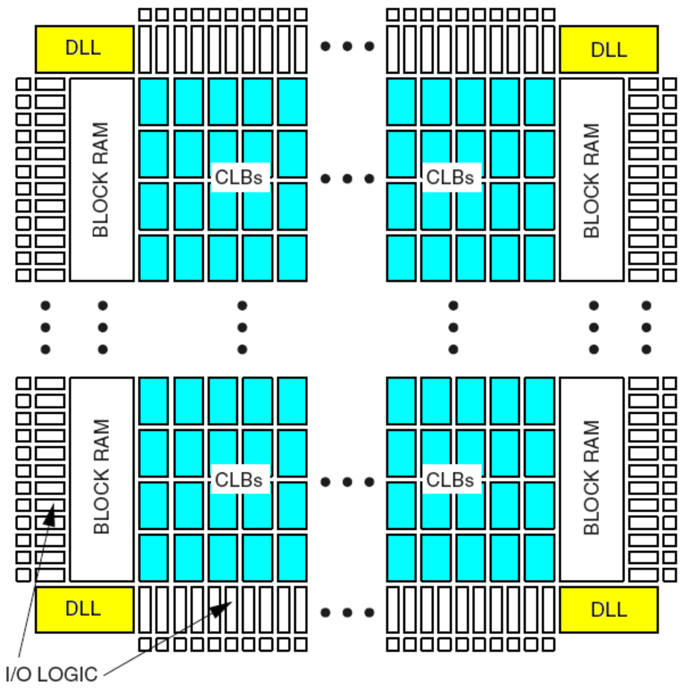

Figure 2.1: A schematic diagram of an FPGA (Xilinx, 2008), The Delay-Locked Loop (DLL) synchronizes the external and internal clocks in an FPGA

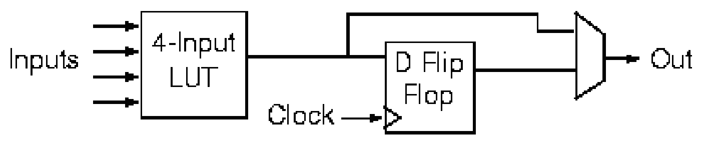

Figure 2.2: A typical Logic Cell (Xilinx, 2008)

2.1.1 Soft-cores, IP-cores

Some configurations, mainly ASICs, Glue Logic and Computing engines, are available via open source or are for sale. These products can be used as building blocks and are called 'soft-cores' or '*intellectual property* (IP)-cores' and usually require only a common FPGA. Soft-cores can be used in an existing project, in order to speed up development time and to reduce the costs. A soft-core that can be used in this project is a Microblaze soft-core CPU.

Xilinx Microblaze
The MicroBlaze core is a 32-bit RISC Harvard architecture soft processor core with a rich instruction set optimized for embedded applications. As a soft-core processor, a Microblaze is implemented entirely in the general-purpose memory and logic fabric of Xilinx FPGAs. A Microblaze is capable of running a real-time Linux version without MMU such as μClinux (Arcturus Networks, 2009) and FreeRTOS (Barry, 2009). This makes it possible to run ANSI C (Kernighan and Ritchie, 1988) code on it which can be produced by 20-sim and gSCP.

2.1.2 Hard-cores

Certain FPGAs embed other hardware on the FPGA chip, i.e. PowerPCs and Digital Signal Processors (DSPs), which can be addressed using logic busses and registers on the FPGA. This way, a designer can use both FPGA and CPU/DSP specific techniques to obtain the best results for the application at hand. Hard-core solutions require less energy and achieve higher clock speeds than soft-core solutions, however, they are more expensive than soft-core solutions.

PowerPC
PowerPC (short for Power Performance Computing, often abbreviated as PPC) is a RISC architecture created by the 1991 Apple-IBM-Motorola alliance, known as AIM. It has all the functionality (e.g. run ANSI C code), and more, that a Microblaze can provide, but then already made in dedicated hardware blocks. It also has MMU support which allows it to run any real-time Linux version such as LynxOS (LynuxWorks Inc, 2009), VxWorks (Wind River Systems, 2009) and QNX (QNX Software Systems, 2009).

2.2 Handel-C

Handel-C is a programming language and Hardware Description Language (HDL) for compiling programs into hardware images of FPGAs or ASICs. It is a rich subset of ANSI C, with non-standard extensions to control hardware instantiation and parallelism.

The subset of C includes all common C language features necessary to describe complex algorithms. Like many embedded C compilers, floating-point data types were omitted. However, floating-point arithmetic is supported through external libraries.

In order to facilitate a way to describe parallel behavior some of the CSP Hoare (1985) keywords are used, along with the general file structure of occam (1984).

2.3 Production Cell

The Production Cell setup by van den Berg (2006) was designed to resemble a real machine consisting of several devices that operate in parallel. It also had to be suitable for distributed control. Furthermore, the setup had to allow for safety implementations, but failures in this safety or the controller should not result in any mechanical damage. Ultimately, the setup should serve as a demonstrator.

The setup consists of 6 axis that operate simultaneously and need to synchronise to pass along metal blocks. These axis are called Production Cell Units (PCUs). Each PCU is named after its dominant function in the system, shown in figure 2.3.

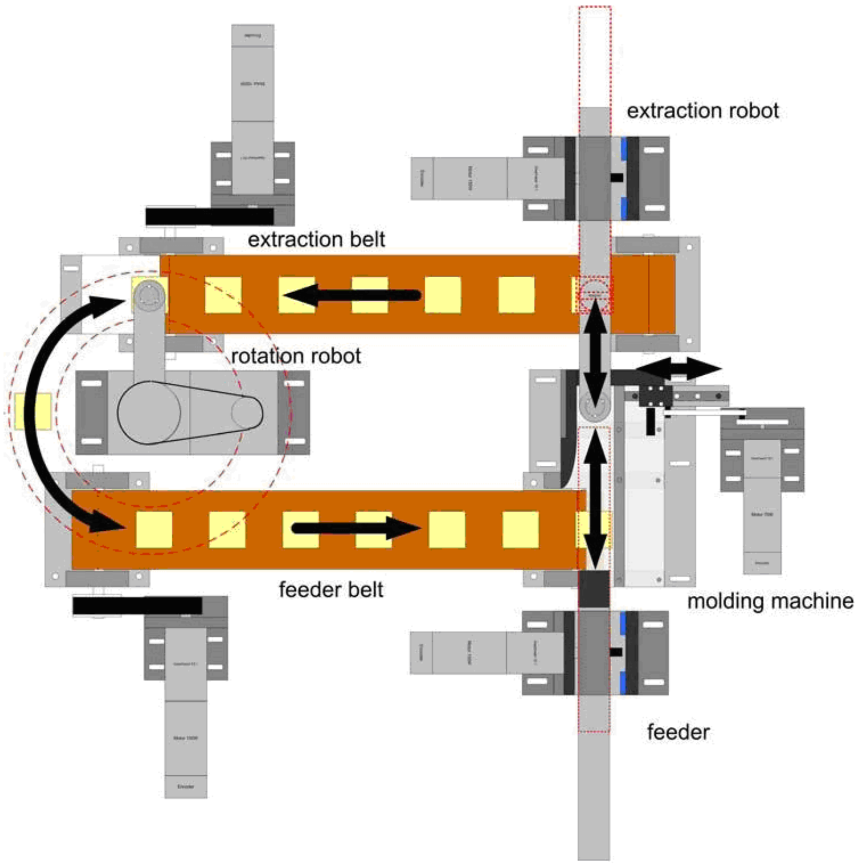

Figure 2.3: The Production Cell setup

The operation sequence begins by inserting a metal block on the feeder belt. This causes the feeder belt to transport the block to the feeder which, in turn, pushes the block against the closed molder door. At this point, the actual molding would take place. After this, the feeder retracts and the molder door opens. This frees the way for the extraction robot, which can now extract the block from the molder. The block is placed on the extraction belt, which transports it to the rotation robot. Finally, the rotation robot picks up the block from the extraction belt and puts it on the feeder belt. Now the cycle starts again.

The belts allow for multiple blocks to be buffered, such that every PCU can be provided with a block at all times, allowing the setup to operate all axis simultaneously. The blocks are picked up using an electromagnet at the end of the extraction robot and the rotation robot.

2.3.1 Previous Work

Several other software-based solutions have been made to control the production cell, as shown in table 2.1.

Who	Method	Loop-controller design	Processor
van den Berg (2006)	Time table based	20-sim	CPU
Maljaars (2006)	gCSP	20-sim	CPU
Huang et al. (2007)	POOSL	20-sim	CPU
Orlic (2007)	SystemCSP	Not implemented	CPU
van Zuijlen (2008)	Handel-C	20-sim	FPGA

Table 2.1: Currently existing controllers

All controllers except the one of van Zuijlen (2008) in table 2.1 are CPU based and therefore use floating-point arithmetic. Van den Berg (2006) uses his own loop controllers, Huang et al. (2007) use the loop controllers from Maljaars (2006), Orlic (2007) has not implemented loop controllers and van Zuijlen (2008) made an integer version for an FPGA architecture of Maljaars (2006) loop controllers. The important differences, however, lie within the design of the software structure.

2.4 Floating-point

Real numbers can be represented on computers in many different ways. One of the commonly used representations is floating-point. A floating-point value is basically a scientific notation of a real number.

Floating-point numbers have three basic components: the sign, the exponent, and the mantissa. The mantissa is composed of the fraction and an implicit leading digit. The exponent base (2) is implicit and need not be stored.

The exponent is biased by (2^{e-1}) - 1, where e is the number of bits used for the exponent field (e.g. if e=8, then (2^{8-1}) - 1 = 128 - 1 = 127). Biasing is done because exponents must be signed values in order to be able to represent both tiny and huge values, but two's complement, the usual representation for signed values, would make comparison for people due to historical reasons harder. To solve this, the exponent is biased before being stored by adjusting its value to put it within an unsigned range suitable for comparison.

For example, to represent a number which has exponent of 17 in an exponent field 8 bits wide: exponent field = 17 + (2^{8-1}) - 1 = 17 + 128 - 1 = 144.

The most significant bit of the mantissa is determined by the value of biased exponent. If 0 < exponent < 2^{e-1}, the most significant bit of the mantissa is 1, and the number is said to be normalized. If exponent is 0 and mantissa is not 0, the most significant bit of the mantissa is 0 and the number is said to be de-normalized. Three other special cases arise:

1 if exponent is 0 and mantissa is 0, the number is ±0 (depending on the sign bit)
2 if exponent = 2^{e-1} and mantissa is 0, the number is ±infinity (again depending on the sign bit)
3 if exponent = 2^{e-1} and mantissa is not 0, the number being represented is not a number (NaN)

A overview of these rules are shown in table 2.2.

Type	Exponent	Mantissa
Zeroes	0	0
Denormalized numbers	0	non zero
Normalized numbers	1 to 2^{e-1}	any
Infinities	2^{e-1}	0
NaNs	2^{e-1}	non zero

Table 2.2: Bias cases Kahan (1997)

Because the MSB of the mantissa can be deducted according to the cases described above, it can be omitted, saving 1 bit. For normalized numbers, which are the most common, the exponent is the biased exponent and mantissa is the mantissa without the most significant bit. A binary value v is represented in floating-point according to the following formula: v = s × 2^e × m

Where:
s = +1 (positive numbers and +0) when the sign bit is 0
s = -1 (negative numbers and -0) when the sign bit is 1
e = exponent - 127 -> 8 bit exponent (in other words the exponent is stored with 127 added to it, also called "biased with 127")
m = 1.mantissa in binary (that is, the mantissa is the binary number 1 followed by the radix point followed by the binary bits of the mantissa). Therefore, $1 \leq m < 2$.

Table 2.3 shows an floating-point 32-bit example of a normalized and de-normalized value.

type	v = s × 2^e × m	s	e	m	sign	exponent	mantissa
Normalized	0.15625	+1	124-127 = -3	1.25	0	01111100	1.01 -> 010...0
De-normalized	2.938736×10^{-39}	+1	0-127 = -127	0.5	0	00000000	01 -> 010...0

Table 2.3: Floating-point representation of a normalized and de-normalized number with, v,s,e,m in decimal and sign, exponent and mantissa in binary

Table 2.4 shows the layout for 32-bit and 16-bit precision floating-point values. The number of bits for each field are shown (bit ranges are in square brackets).

Precision	Sign	Exponent	Mantissa	Bias
single precision 32-bit	1 [31]	8 [30-23]	23 [22-0]	127
half precision 16-bit	1 [15]	5 [14-10]	10 [9-0]	15

Table 2.4: Floating-point layout

For 16-bit floating-point apply the same rules as for 32-bit floating-point. The difference between 32-bit and 16-bit floating-point is the range of the decimal numbers that can be represented. 32-Bit floating-point can represent decimal numbers from 1.175×10^{-38} up to 3.4×10^{38} while 16-bit floating-point can only represent decimal numbers from 2.98×10^{-8} up to 65504.

All rules and specifications for floating-point are specified in IEEE 754 (2008) standard.

ANSI C Float to Handel-C Floating-point

The following example shows how the declaration of 2 float values in ANSI C and in Handel-C look like.

In ANSI C

```
float a = 0.007;
float b = -0.31;
```

In Handel-C

```
// Float structure : {sign, exponent, mantissa}
Float a = {0, 119, 6643777 }; //  0.007 written in Handel-C floating-point
Float b = {1, 125, 2013265 }; // -0.31  written in Handel-C floating-point
```

The Handel-C floating-point structure contains the integer representation of the sign, exponent and mantissa separated by commas.

For converting float values to the Handel-C floating-point structure a tool made by Schmidt (2009) is used. This tool can convert a float value to the binary representation. From this binary representation the integer representation of the sign, exponent and mantissa can be derived.

2.5 Fixed-point

A fixed-point number represents a number consisting of a integer part (e.g. 5) and a fractional part (e.g. .375). The format of a fixed-point number is described as "m.n", where m is the number of bits in the integer part and the n is the number of bits in the fractional part. m + n is the total number of bits. For example, "3.4" (see figure 2.4) means a 8-bit integer with an 3-bit integer part and a 4-bit fractional part. The first bit of the integer part is noted as the sign bit. As shown is figure 2.4 the binary 01100110 gives the decimal number 5.375.

Because fixed-point operations can produce results that have more bits than the operands, information loss may occur. For instance, the result of fixed-point multiplication could potentially have as many bits as the sum of the number of bits in the two operands. In order to fit the result into the same number of bits as the operands, the answer must be rounded or truncated. If this is the case, the choice of which bits to keep is important.

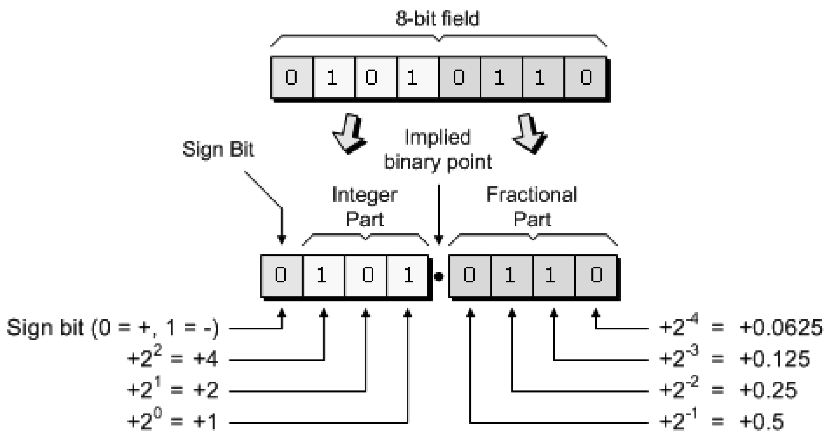

Figure 2.4: fixed-point representation of the decimal number 5.375

3 Loop Controller Implementation Alternatives

This chapter points out, how alternative possibilities for implementing loop controllers; floating-point, fixed-point, integer, soft-core and hard-core (table 1.1), will fit in the current work flow of embedded control applications, and which one is used to implement the Production Cell with.

Section 3.1 elaborates the current design flow and how the loop controllers will fit into this. Section 3.2 contains the implementation of the floating-point loop controller. Section 3.3 contains the implementation of the fixed-point loop controller. Section 3.4 contains the integer, soft-core and hard-core loop controller implementations. Section 3.5 contains information about the used Motion Profile (set point generator). Finally in section 3.6 the results of all loop controllers are discussed and a loop controller is chosen to implement the Production Cell with.

3.1 Design Flow

Figure 3.1 (same as figure 1.1) shows the design flow for embedded control applications. The flow starts with creating a 20-sim of the physical system and a gCSP model of the system communication structure. The second part of the design flow is code generation. From each part of the 20-sim or gCSP model, code can be generated. In case of this project, code generation of the loop controller part. The last step of the design flow is running the generated code on the target computer. The ANSI C generated code can be run on the target computer with the use of 20-sim 4C and the Handel-C generated code can be used to configure the FPGA chip.

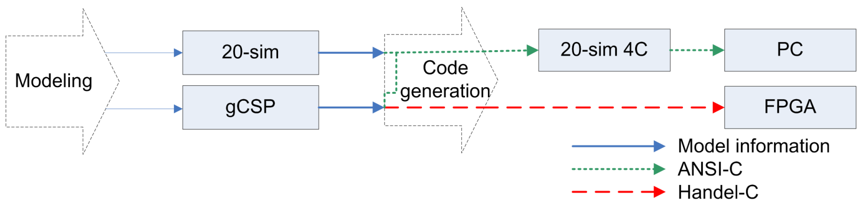

Figure 3.1: Design flow for embedded control applications adapted from (van Zuijlen, 2008)

Currently, 20-sim cannot generate code for an FPGA architecture. gCSP can generate Handel-C code for an FPGA architecture, however, this is only limited to basic (channels, par, seq, ..) Handel-C constructions and only for the data type integer. For the data type floating-point and fixed-point, code generation still needs to be done manually which is time consuming.

The design effort for the Production Cell can be optimized in different parts of the design flow. Van Zuijlen (2008) optimized the design effort in the *code generation* part of the design flow by using integer as data type for the loop controller. However, the choice for integer as data type, resulted in redesigning the existing 20-sim Production Cell loop controller model. This redesign was needed because the existing 20-sim Production Cell loop controller model was designed for the float data type. Optimizing on code generation resulted in 4 weeks of design effort for one person.

Another way to optimize design effort is on the control part. This can be done by using the existing 20-sim Production Cell loop controller. This requires code generation for the float data type. In order to generate code for the float data type an interface needs to be made, that can convert float to the floating-point Handel-C structure. By optimizing on the control part a floating-point loop controller implemented in Handel-C would be required.

The goal of the new loop controller implementation regarding the design flow, is to fit in with the least amount of design effort for a model based design flow. In the next 3 sections alternative loop controller implementations are discussed.

3.2 Floating-point Loop Controller Implementation

Designing a floating-point loop controller involves a lot of implementation choices. Figure 3.2 shows all implementation choices for the floating-point loop controller.

Accuracy

For a floating-point data type different bit accuracies are possible. The bit accuracies that are used during this project are 16-bit and 32-bit. The 32-bit accuracy is used because Handel-C offers support libraries for 32-bit floating-point calculations. The 16-bit accuracy, for which manually a support library needs to be made with the Xilinx Coregen tool, is used to check how much FPGA-resource usage can be saved compared to the 32-bit accuracy.

Language

Van Zuijlen (2007) stated that Handel-C is a good choice as hardware description language for embedded control applications. Therefore, Handel-C is used during this project to implement the loop controller for an FPGA architecture.

Support Library

The support library provides functions that can perform actions on floating-point numbers. There are 2 versions of the support library for Handel-C. One for performing sequential floating-point calculations and one for performing pipelined floating-point calculations. The difference between the sequential and pipelined floating-point support library is that the pipelined version is faster and using more FPGA-resources than the sequential version.

Arithmetic Operation Wrapper

In order to use functions from the support library a arithmetic operation wrapper is made that combines all function calls that are needed to do one arithmetic operation (multiply, add, subtract and divide) calculation on a floating-point number.

For example, doing a multiplication between the numbers x en y. In the loop controller Handel-C code this looks like:

```
result = doMult(x,y); //self made wrapper function
```

With the *doMult* from arithmetic operation wrapper, contains the following implementation code:

```
Float doMult(Float op1,Float op2) { // Handel-C implementation
    FloatPipeCoreSet(&MyMultiplier, op1, op2);
    while(FloatPipeCoreResult (&MyMultiplier) ==0)
     delay;  //wait until finished
    FloatPipeCoreGet(&MyMultiplier, &result);
    return result;
}
```

An advantage of this self made wrapper is that 20-sim code generation of floating-point calculations is possible now. For example, the calculation $z = x \times y$ in 20-sim is replaced by $z =$ *doMult(x,y)* in Handel-C.

In this function, the functions *FloatPipeCoreSet*, *FloatPipeCoreResult* and *FloatPipeCoreGet* from the Handel-C floating-point support library are used.

The Xilinx Coregen tool does not include the option to select different methods (sequential or pipelined) for floating-point calculations. Therefore, the 16-bit support library has only one version to do floating-point calculations. To integrate the 16-bit floating-point support library with Handel-C, a wrapper is needed to connect the library to Handel-C. This version also requires an additional floating-point arithmetic operation wrapper.

PCU Execution Order

For this project the loop controller needs to control motors in the Production Cell. The Production Cell has 6 PCUs which means, controlling 6 motors. These motors can be executed in 2 different ways.

- sequential, 1 loop controller function that is reused 6 times to control 1 motor at a time.
- parallel, 6 loop controllers functions, which each control 1 motor.

All these choices give a total of 4 floating-point 32-bit implementations and 2 floating-point 16-bit implementations which are discussed next.

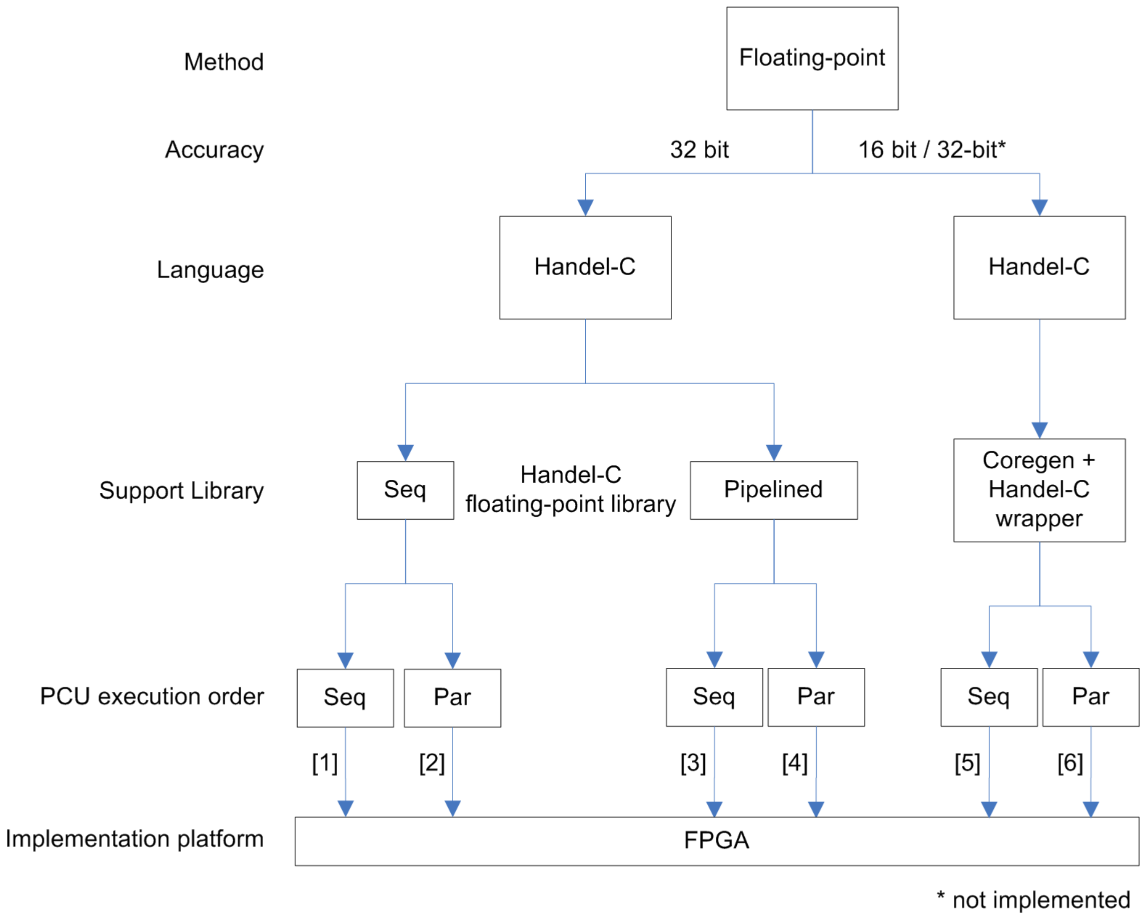

Figure 3.2: Floating-point implementation choices

3.2.1 Floating-point 32-bit Loop Controller Implementation

Floating-point 32-bit has 4 different implementation routes (figure 3.2):

- route [1] sequential floating-point support library with sequential PCU execution.
- route [2] sequential floating-point support library with parallel PCU execution.
- route [3] pipelined floating-point support library with sequential PCU execution.
- route [4] the pipelined floating-point support library with parallel PCU execution.

Route [1] and [3] use sequential PCU execution which means that every motor is controlled after another. Executing the PCUs in this way results in the possibility to reuse the loop controller function and thus saving FPGA-resources.

Route [2] and [4] use parallel PCU execution which means that every motor is controlled at the same time. Executing the PCUs in this way means that each PCU needs to have its own loop controller function resulting in an FPGA-resource usage explosion.

Because the existing Production Cell loop controller model works with float values no redesign of the loop controller model is needed to let the floating-point loop controller work. This makes floating-point 32-bit suitable to integrate into the design flow. However, when reusing the existing loop controller the internal 20-sim resolution, which is 64 bit, must be taken into account when testing and comparing results.

Implementation Detail

The implementation of the floating-point 32-bit loop controller consists of basic Handel-C constructions. If the float to Handel-C floating-point structure (see section 2.4) conversion is implemented in 20-sim and gCSP, code generation of the loop controller is possible.

Before the loop controller can end, the error feedback to the controller needs to be at least 0.5 mm or less (see section 1.3.3). If this is not the case the loop controller keeps controlling the motor until this minimum error is reached.

Results & Validation

The results of the floating-point loop controller implementations are validated by comparing 20-sim simulation outputs with the outputs from the FPGA. Both outputs are based on an error of 0 for the PID algorithm. For validation, the results of the Production Cell rotation arm are used. The loop controller controls the motors by sending a Pulse-Width Modulation (PWM) signal to them. For the motors of the Production Cell this PWM value lies between -2047 and 2047. This corresponds with 100% motor speed to the left(-2047) and 100% motor speed to the right(2047).

The 20-sim simulation outputs are stored with a resolution of 10^{-15}. The FPGA output is limited to a resolution of 10^{-5}. This is a restriction of the self made tool that converts the Handel-C floating-point structure to the float data type (see Appendix A). As result of this resolution difference, a small deviation between the 20-sim and FPGA results is possible.

In figure 3.3 the first graph shows the results of the 20-sim output the second graph the FPGA output and the third graph shows the difference between the 20-sim output and the FPGA output. As can be seen the difference between the 2 output is less than 2%. This difference is caused due to the resolution difference and, therefore, is neglectable showing that the floating-point loop controller works.

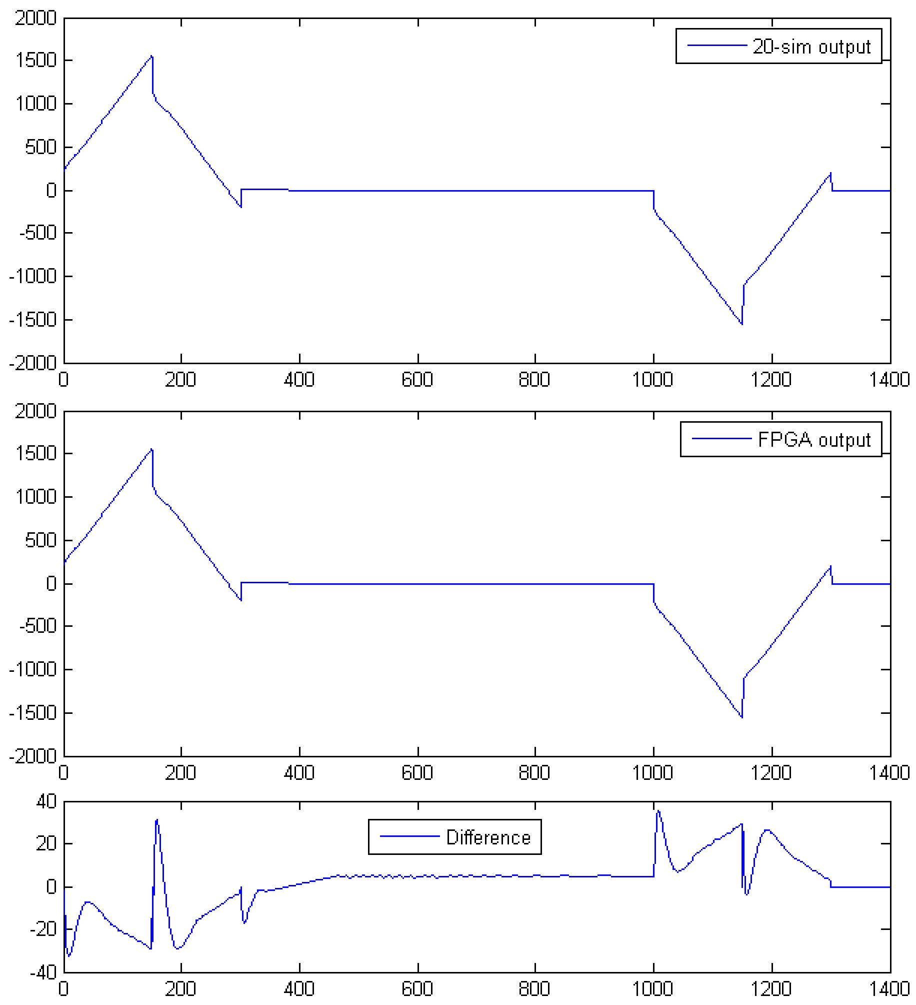

Figure 3.3: Floating-point 32-bit loop controller output results, x in ms and y in PWM pules, note the y scale on the third graph

3.2.2 Floating-point 16-bit Loop Controller Implementation

Floating-point 16-bit has 2 different implementation routes (figure 3.2):

- route [5] sequential PCU execution.
- route [6] parallel PCU execution.

For floating-point 16-bit only one floating-point support library is available, which is made in Xilinx Coregen. Comparing to 32-bit floating-point, the 16-bit version should be smaller in FPGA-resource usage. To make 16-bit floating-point work for the Production Cell system the standard layout (1 sign, 5 exponent and 10 mantissa bits) need to be changed. As pointed out in the floating-point section in chapter 2, 16-bit floating-point can only go up to a value of 65504. In the current loop controller model of the Production Cell certain values exceed this value. Therefore, an adjustment of the 16-bit floating-point layout is needed. The new layout will be 1 sign, 6 exponent and 9 mantissa bits (range 1.82×10^{-12} up to 4.290.772.992). With this adjustment some precision gets lost but a wider range of numbers can be obtained.

The 16-bit floating-point loop controller requires a conversion from a float value to the new declared 16-bit floating-point structure. Because this structure deviates from the standard structure no tool exist to do this conversion. This results in more design effort to integrate a floating-point 16-bit loop controller in the design flow than the floating-point 32-bit loop controller. Due to this reason route [5] and [6] are not implemented. In order to tell something about FPGA-resource usage of the 16-bit floating-point only the support library is made and compared to the 32-bit floating-point one.

3.3 Fixed-point Loop Controller Implementation

Fixed-point has 2 different implementation routes (figure 3.4):

- route [1] is based on sequential PCU execution.
- route [2] is based on parallel PCU execution.

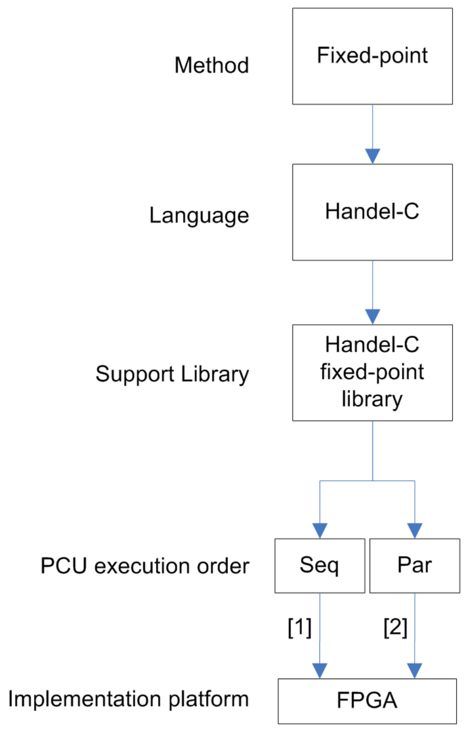

Figure 3.4: Fixed-point implementation possibilities

Route [2] will be faster in calculation time than route [1] but will also consume more FPGA-resources because of the parallel structure. The FPGA-resource usage for fixed-point will be a bit higher than the integer implementation and much lower than the floating-point implementation as can be seen in section 3.6.

To implement a fixed-point loop controller in the design flow, the current 20-sim and gSCP loop controller model need to be adjusted to the fixed-point data type. The biggest challenge is to find a suitable fixed-point layout with the right width for the integer and fraction part. Because the values in the Production Cell models have a range of really small numbers to big numbers fixed-point is not an ideal format. For the small numbers a small sized integer width is needed and a big sized fraction width. And for a big number, it is the other way around. Accuracy problems occur when big numbers and small numbers come together. Accuracy can get lost due to rounding errors when the result does not fit in the specified fixed-point layout.

Due to this drawback only a motion profile generator is made using fixed-point with a size of 16 bit, 8 bit for the integer part and 8 bit for the fraction part. This was done to get a FPGA-resource estimation of fixed-point compared to floating-point 32-bit. See results of this in section 3.6.

3.4 Other Loop Controller Implementations

Integer Loop Controller

The integer loop controller implementation was made by van Zuijlen (2008) which used a parallel PCU execution order. The results of this project will be compared to the integer implementation.

CPU core in FPGA

- soft-core CPU with floating-point unit loop controller

  The soft-core CPU with floating-point unit loop controller can use a Microblaze to run the loop controller. The Microblaze cost about 1800 LUTs of FPGA-resources resulting in 1800 LUTs less available for the loop controller.

- hard-core CPU with floating-point unit loop controller

  The hard-core CPU with floating-point unit loop controller can use a PowerPC to run the loop controller. The PowerPC uses dedicated hardware, saving FPGA-resources.

An advantage of these solutions is that the existing software loop controller models can be reused, saving design effort.

Drawbacks from the hard-core and soft-core solutions are:
- The design becomes complicated due to the interface between the hard-core and soft-core CPU software and Handel-C software when reusing the existing software loop controllers
- The CPU needs a scheduler when all 6 PCUs run on the same soft-core or hard-core CPU.

Both soft-core and hard-core solutions are not tested during this project due to the fact that no FPGA was operational to do this. In the future these solutions need further research and testing. A suitable FPGA for these solutions is the Xilinx ML510. This FPGA has 2 PowerPCs together with an Virtex 5 FX130T FPGA. Both solutions can be implemented on this FPGA.

3.5 Motion Profile Generation for the Loop Controller

A part of the loop controller is the motion profile generator which provides the set points. The motion profile generates 3 signals: acceleration,velocity and position. Van Zuijlen (2008) suggested that when floating-points are used in a Xilinx Spartan II FPGA, a 48 times bigger FPGA is needed. However, when looking at the 3 generated signals in the motion profile only the acceleration signal is needed. The velocity and position signal can be derived from that. The velocity is obtained by integrating the acceleration, the position is obtained by integrating the velocity. This method of generating the signals saves a lot of space in the FPGA and makes it possible to fit a motion profile in the used FPGA. Furthermore, it gives the possibility to compute the set points during run-time.

Motion Profile Implementation

The motion profile is implemented with a symmetrical acceleration profile. This means that during the runtime of the motion profile the acceleration parameter cannot change and all motion profiles, that can be used by the loop controller need to have the same shape as in figure 3.5.

The implementation of the motion profile in Handel-C is done by using the following formulas:
- for velocity: $velocity = velocity + acceleration \times sampletime.$
- for position: $position = position + velocity \times sampletime.$

Where the sampletime is 0.001.

When a non symmetrical acceleration profile is used in this motion profile implementation, the end position is not the same as the start position resulting in a position difference which can cause unexpected behavior in the system.

For example, the rotation arm stops at 90 degrees were it should be at 0 degrees. In order to pick up the block the rotation arm has to turn 90 degrees to get to the 0 degrees start position, then pick up the block and turn it another 180 degrees to move the block to the other belt. This all has to be done in the same time as the normal 180 degrees movement. This results in a situation where the movement of the arm is too fast for the magnet to keep the metal block attached to the arm, causing the metal block to fall of the rotation arm.

Validation

To validate this method, the results of the motion profile signals (acceleration,velocity and position) generated by the 20-sim simulation are compared to the FPGA results.

In figure 3.5 the results of the acceleration signal are shown. The first graph shows the 20-sim results, the second graph shows the FPGA results and the third one shows the difference between them. The difference between them is exactly 0, this was also expected from a predefined fixed signal.

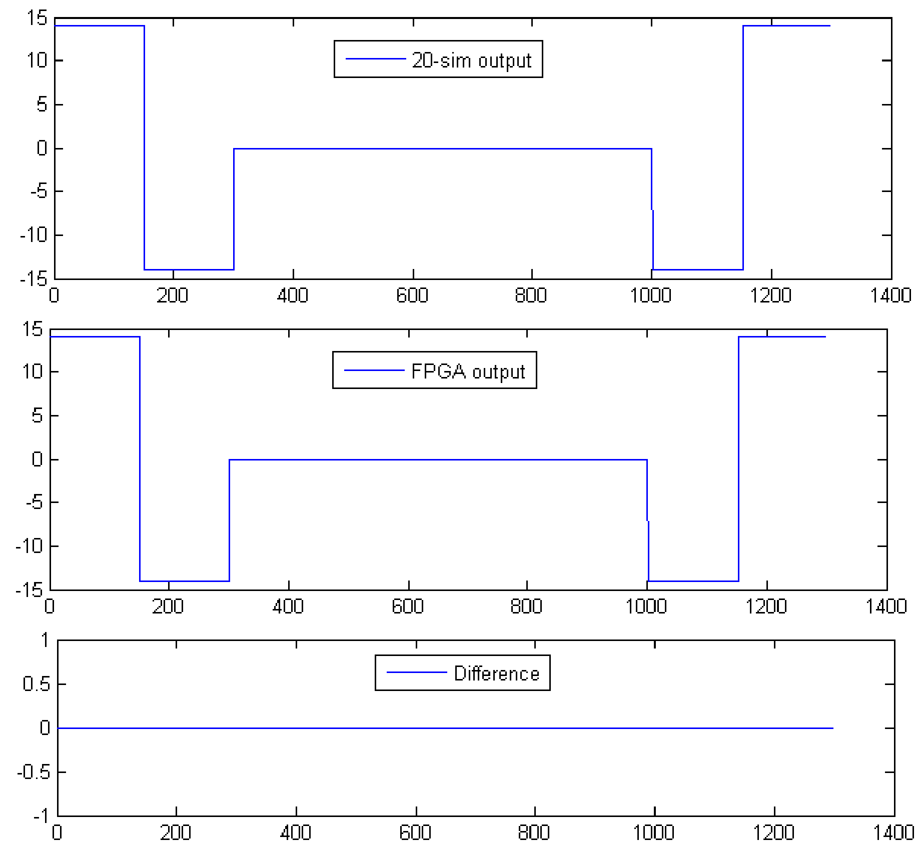

Figure 3.5: Acceleration signal for the rotation arm with the x scale in ms and y scale in m/s^2

In figure 3.6 the results of the velocity signal are shown. The first graph shows the 20-sim results, the second graph shows the FPGA results and the third one shows the difference between them. As can be seen the signals are almost the same. The difference is too small to have any influence on the end result of the loop controller, therefore, it is neglectable.

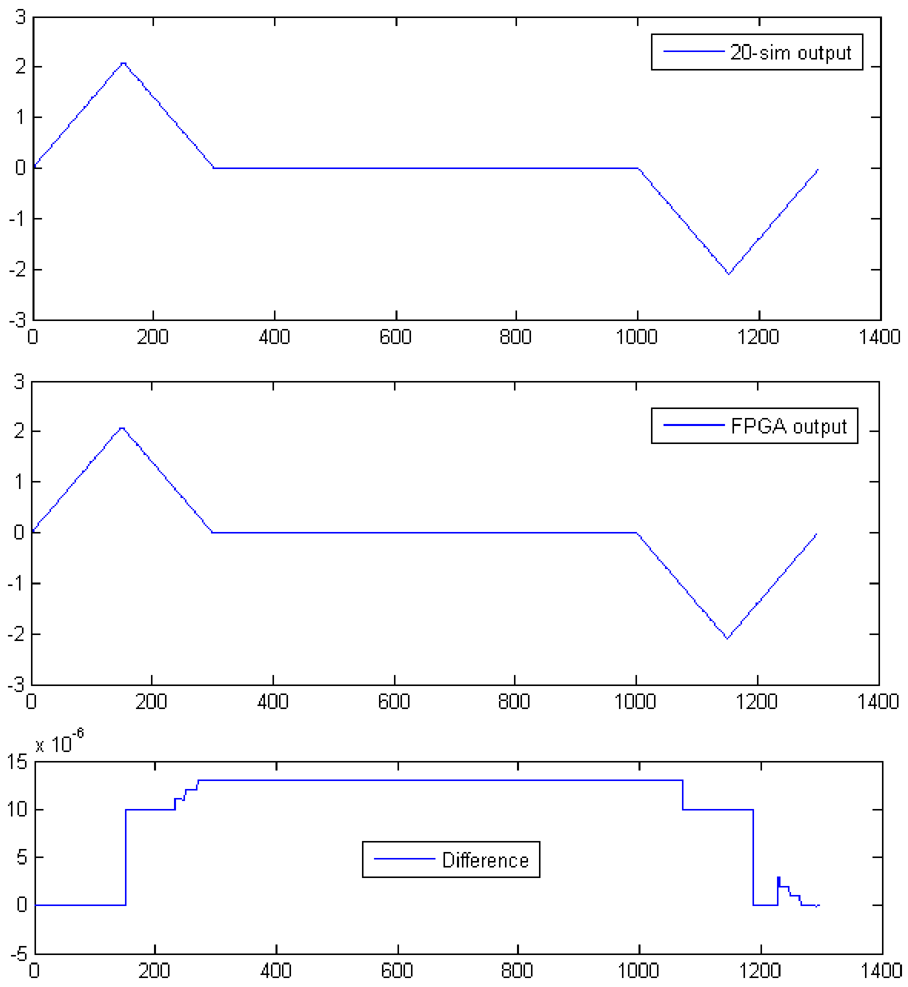

Figure 3.6: Velocity signal for the rotation arm with the x scale in ms and y scale in m/s, note the y scale of the bottom graph

In figure 3.7 the results of the position signal are shown. The first graph shows the 20-sim results, the second graph shows the FPGA results and the third one shows the difference between them. As can be seen the difference slightly increases during run-time of a single motion profile, with a new run the difference starts at 0 again. The difference is too small to have any influence on the loop controller results. Therefore, it is neglectable.

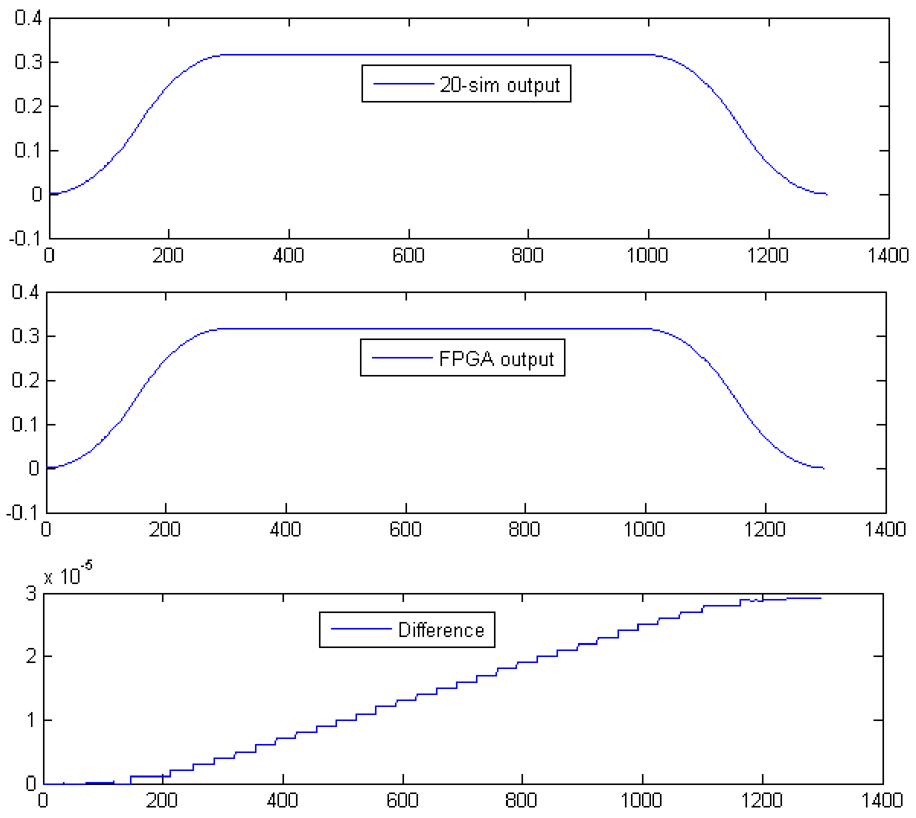

Figure 3.7: Position signal for the rotation arm with the x scale in ms and y scale in m, note the y scale of the bottom graph

When using a non symmetrical acceleration profile the integration method implemented for the floating-point loop controller can not be used. In order to work with a symmetrical acceleration profile, the signals need to be generated in advance and stored into memory(blockram) on the FPGA, this can be done by generate all signals with 20-sim and then covert all float values to the Handel-C structure and import these values in Handel-C. A downside of this method is, that it requires memory from the FPGA. In case of the Spartan 3 FPGA, 589k memory is available. When dealing with 32-bit floating-point values each floating-point value takes up 38 bit of memory consisting of 32 bit for the value and 6 administration bits. In total the Spartan 3 FPGA can store 589824/38 = 15521 (maximum of 15 seconds with 1000 samples per second) set points assuming other components do not use memory.

3.6 Results

In order to choose the best floating-point loop controller the choice between 16-bit and 32-bit needs to be made. Table 3.1 shows the FPGA-resource usage of the 4 arithmetic operations needed by the loop controller in the designated format. The difference between the 16-bit and 32-bit FPGA-resource is surprisingly small. Possibly, this can be explained due to the fact that the 16-bit operations are fully IEEE 754 compliant and the 32-bit version not. With this small difference and the fact that Handel-C already offers support libraries for 32-bit floating point, floating-point 32-bit is the best option to choose.

	16-bit	**32-bit**
	LUTs	LUTs
Add	264	117
Subtract	263	213
Multiply	165	268
Divide	193	498
Total	885	1096

Table 3.1: Floating-point 16-bit made in Xilinx Coregen vs 32-bit made in Handel-C arithmetic operation FPGA-resource usage

During the implementation of the floating-point 32-bit routes (figure 3.2) all test steps from section 1.3.2 in chapter 1 are followed.

In table 3.2 the results of the floating-point 32-bit routes are shown. Each route is implemented with both motion profile (MP) options, during runtime and in blockram. As can be seen only route [1] with the motion profile during runtime and route [3] with motion profile during runtime will fit on the Spartan 3 FPGA. Route [2] and route [4] exceed the maximum available FPGA-resources due to the fact that each PCU has its own floating-point library. Storing all motion profiles in blockram does not fit. To overcome this problem the number of used set points need to be changed or an FPGA with more memory needs to be used. As expected, the calculation time of all routes is way below the maximum of 1 ms. Therefore, the best route is chosen based on the lowest design effort and FPGA-resource usage.

Figure 3.8 shows the FPGA-resource usage compared to the design effort for all floating-point loop controller routes and the integer loop controller. The design effort for all floating-point loop controller routes are the same. For the floating-point loop controller the estimated design effort is 1 week of work for one person. For the integer loop controller the estimated design effort is 4 weeks of work for one person. As can be seen route [1] is the smallest in FPGA-resource usage. Therefore route [1] will be chosen as the best route of the floating-point 32-bit routes, to use when implementing the loop controller.

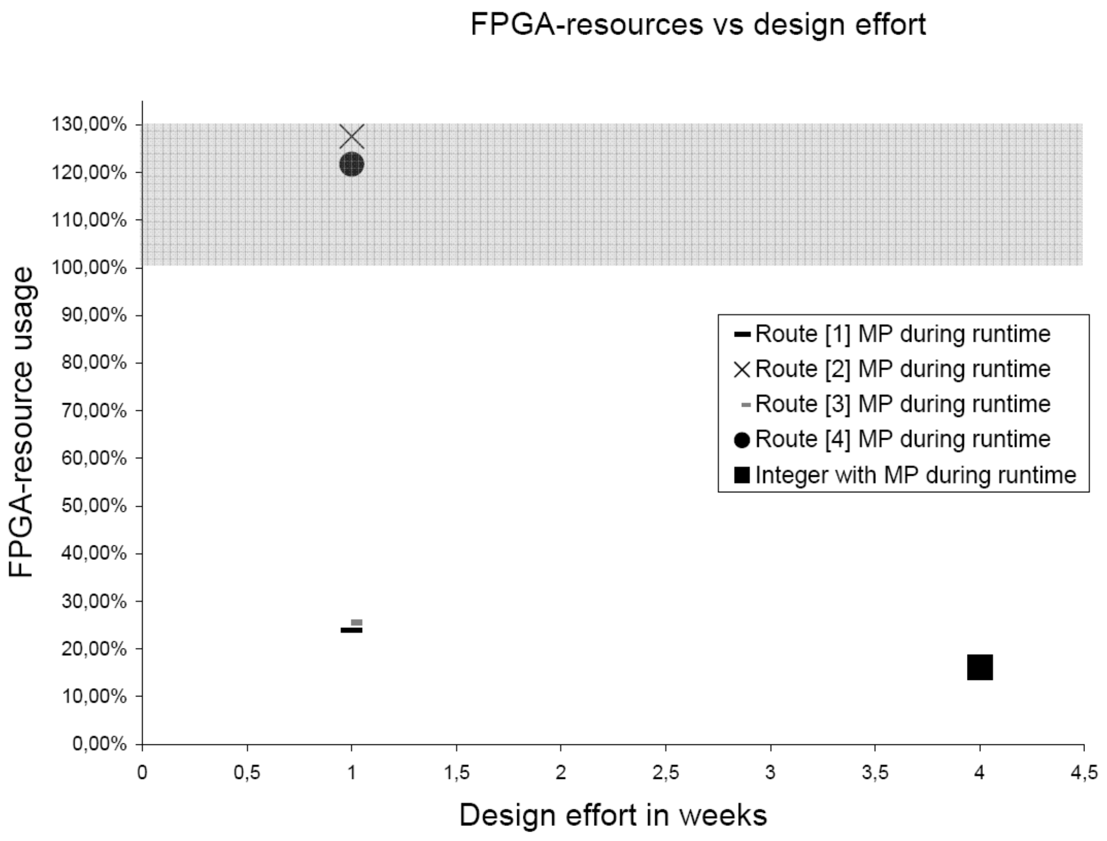

Figure 3.8: FPGA-resource usage vs design effort for the floating-point 32-bit routes and integer loop controller

To get an estimation of fixed-point FPGA-resource usage compared to floating-point only the motion profile generation is made and compared to floating-point 32-bit version. This comparison is shown in table 3.3. The fixed-point version uses a 16-bit format divided in 8 bit for the integer part and 8 bit for the fraction part. As can be seen, the fixed-point version is about factor 10 smaller than the floating-point 32-bit version. The downside of fixed-point is that the current models need to be changed and the layout wide for fixed-point is difficult to choose. In despite of the smaller FPGA-resource usage of fixed-point compared to floating-point, fixed-point will not be chosen as loop controller due to the fact that the emphasis of this project is the design effort which for fixed-point is way higher than for floating-point.

The Integer loop controller made by van Zuijlen (2008) is shown in table 3.4. The floating-point 32-bit implementation compared to the integer one shows that the integer version is about 2000 LUTs smaller and about 90 times faster in calculation time. However, when regarding how complex floating-point calculations are, this difference is not big, especially when the design effort is taken into account. Floating-point requires no changing of current design models while integer does.

After discussing all possibilities, Floating-point is the best suitable data type to use in the loop controller. From the floating-point alternatives, route [1] from figure 3.2 will be used to implement the Production Cell with.

	LUT	**FF**	**Mem**	**ALUs**	**LUT %**	**Mem %**	**Fits**	**C**	**C %**
Maximum	26624		589824		32				
Route [1]									
MP in blockram	5669	3825	671744	4	21,29%	113,89%			
MP during runtime	*6385*	*4531*	*0*	*4*	*23,98%*	*0,00%*	*X*	*41,54 μs*	*4.15 %*
Route [2]									
MP in blockram	32530	21492	868352	24	122,18%	147,22%	-		
MP during runtime	33967	22648	0	24	127,58%	0,00%	-	6.92 μs	0,69%
Route [3]									
MP in blockram	6508	4127	671744	4	24,44%	113,89%			
MP during runtime	*6816*	*4818*	*0*	*4*	*25,60%*	*0,00%*	*X*	*38,41 μs*	*3.84 %*
Route [4]									
MP in blockram	31181	21937	868352	24	117,12%	147,22%	-		
MP during runtime	32407	23792	0	24	121,72%	0,00%	-	6.4 μs	0,64%

Table 3.2: Result of the floating-point 32-bit loop controller implementation routes with FF being a subset of LUTs and C = calculation time

	LUT	**FF**	**Mem**	**ALUs**	**LUT %**	**Mem %**
Fixed-point (8.8)						
MP during runtime	220	58	0	2	0,83%	0,00%
Floating-point 32 bit						
MP during runtime	2785	1338	0	4	10,46%	0,00%

Table 3.3: Floating-point compared to fixed-point motion profile generation

	LUT	**FF**	**Mem**	**ALUs**	**LUT %**	**Mem %**	**Fits**	**C**	**C %**
Maximum	26624		589824	32					
MP during runtime	4316	198	0	0	16,21%	0,00%	X	0.464 μs	0.05 %

Table 3.4: Result of the integer loop controller implementation (all 6 loop controllers with MP) (C = calculation time)

4 Redesign of the Production Cell controller

This chapter elaborates how the current Production Cell framework structure used for the integer loop controller implementation needs to be changed to let the proposed floating-point 32-bit loop controller work (section 4.1). After this, the working of the sequence control (section 4.2), algorithm (section 4.3) and safety layer (section 4.4) part of the Production Cell framework will be elaborated. Finally the results (section 4.5) of the new system will be discussed.

4.1 Structure

The current Production Cell top level structure made by van Zuijlen (2008) is shown in figure 4.1 This structure is designed for distributed control(de-centralized). Hence, this implies that every unit of the setup must contain enough intelligence to operate independently, rather than having a central point of intelligence.

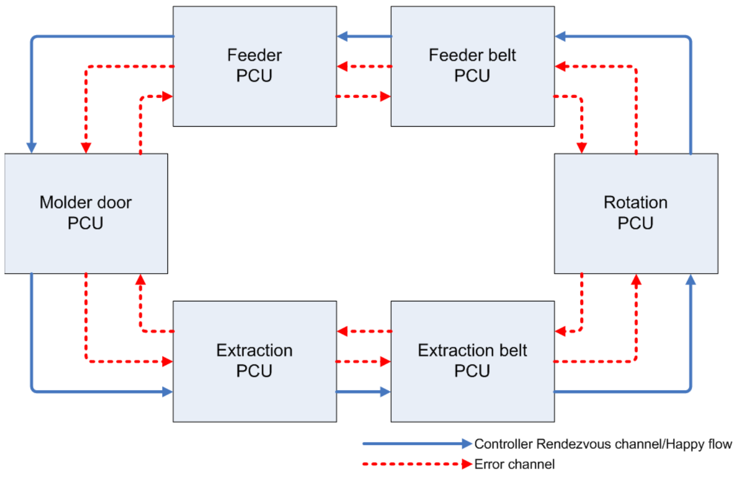

Figure 4.1: Top level design for the integer loop controller made by van Zuijlen (2008)

Every PCU (figure 4.2) is a single entity that has its own algorithm (Controller), sensor, magnets, motor and encoder (low level hardware block), command and safety blocks. For the implementation of the Production Cell with the chosen floating-point loop controller a different model, shown in figure 4.3 (this new structure is a combination of figure 4.1 and 4.2 adjusted for the new loop controller), is more suitable.

The new structure is based on a centralized design due to the fact that each PCU reuses the same loop controller algorithm.

The only problem in the current model is that a sensor belongs to only one PCU. In some cases it is desirable to have a single sensor value available in more PCU blocks, for example the molder door sensor. The feeder PCU needs it to check whether it can place a block inside the molding area (the door needs to be closed) and the extraction PCU needs it to know when the door is open so the block can be extracted. To solve this, a central sensor monitoring block is implemented. Such a block also has another advantage; it can detect a deadlock in the system. Looking at the current safety structure it is not possible to detect a deadlock by the system. Because a PCU only knows information about itself and the previous one, it can never tell if all PCU blocks are blocked.

With a central safety block that can communicate with a central sensor block and algorithm block, deadlock can be detected and the system can react faster on errors. Once an error occurs the general safety can overwrite all state machine blocks at the same time. With the structure that van Zuijlen (2008) made an error signal needs to go from one to another PCU block until all blocks are updated.

The global process gets updated by the safety layer and all state machine blocks can read from it. In the globals process all information of the sensors values are stored into variables. With this new structure a PCU block only contains the state machine of each PCU. Therefore the PCU blocks are renamed to state machine blocks.

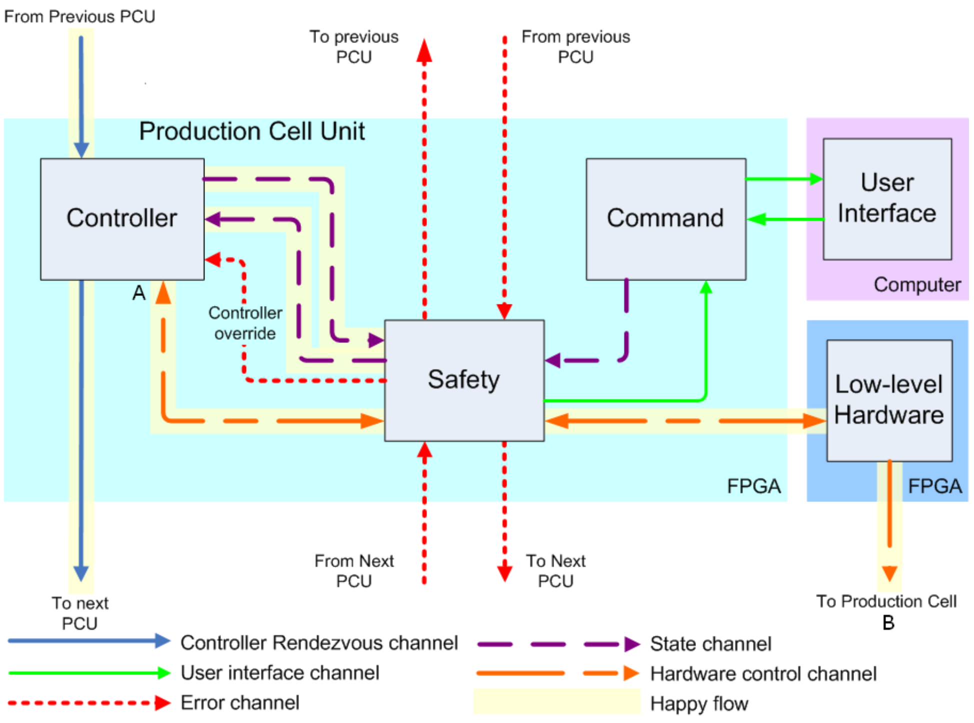

Figure 4.2: Model of a PCU for the integer loop controller made by van Zuijlen (2008)

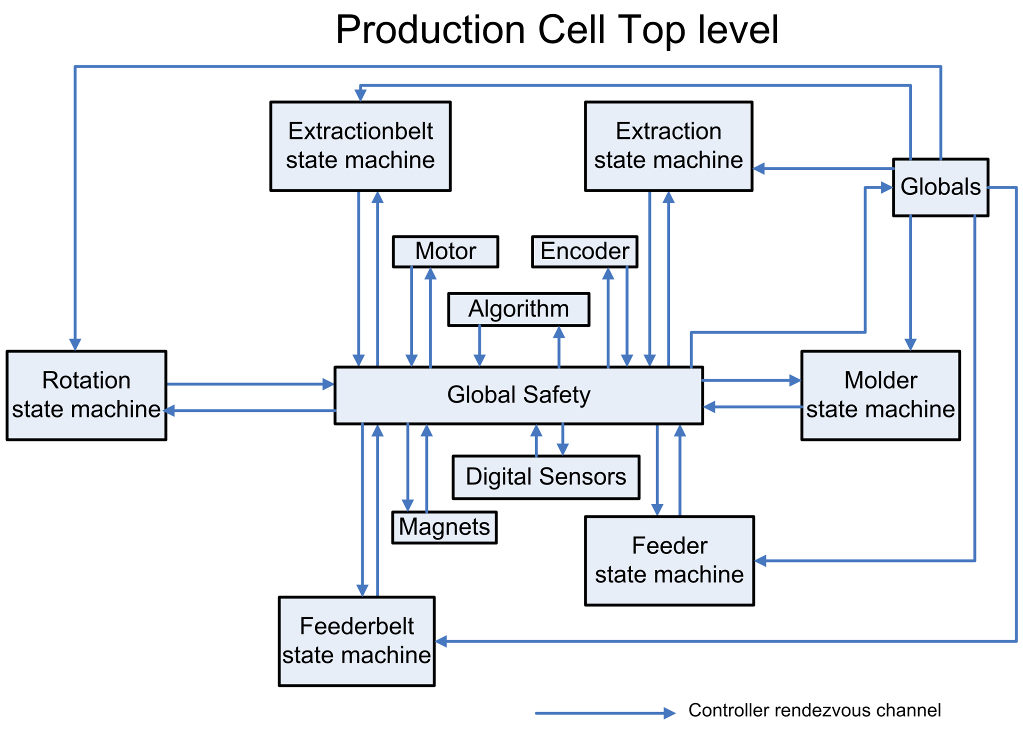

Figure 4.3: Top level design for the new Production Cell controller

4.2 Sequence Control

Actions in the Production Cell System are controlled by a state machine. States used in the Production Cell are: idle, motion profile, run, hold, error, homing and deadlock. The communication between the states for the rotation, feeder, molder and extractor motors is shown in figure 4.4. The communication between the states for the extractionbelt and feederbelt is shown in figure 4.5.

All PCUs start in the homing state. After the homing all PCUs are in the idle state. The rotation, feeder, molder and extractor state machine can go from the idle state to the motion profile, error and deadlock state. From the motion profile state, the error and idle state can be activated. When the deadlock state is reached, the system will wait until a buffer position within the Production Cell is released. After that the state will change to idle. Once the system reaches the error state no state change is possible anymore and the system stops.

For the extractionbelt and feederbelt 2 extra states (run and hold) are added compared to the other PCUs. This is done because the belts either are running or stand still. This means that the error and deadlock state can be reached from the hold and run state also. From the idle state the run and hold states can be reached through the motion profile state.

All state changes are initiated by changes in sensor values or error signals in the system.

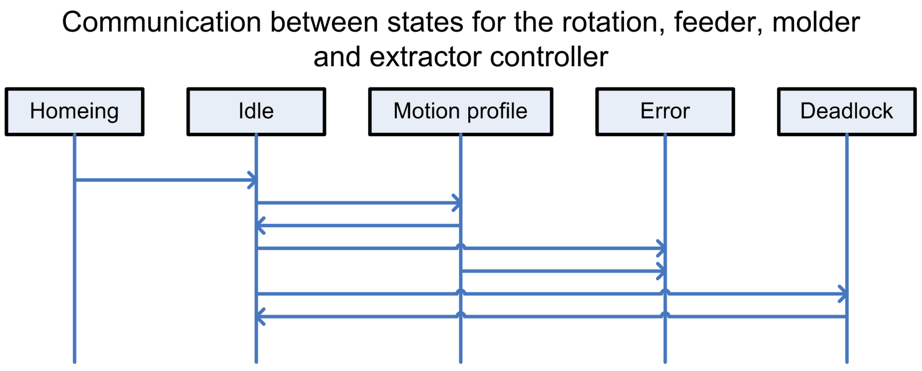
Communication between states for the rotation, feeder, molder and extractor controller
Homeing
Idle
Motion profile
Error
Deadlock

Figure 4.4: Communication between states for the rotation, feeder, molder and extractor controller

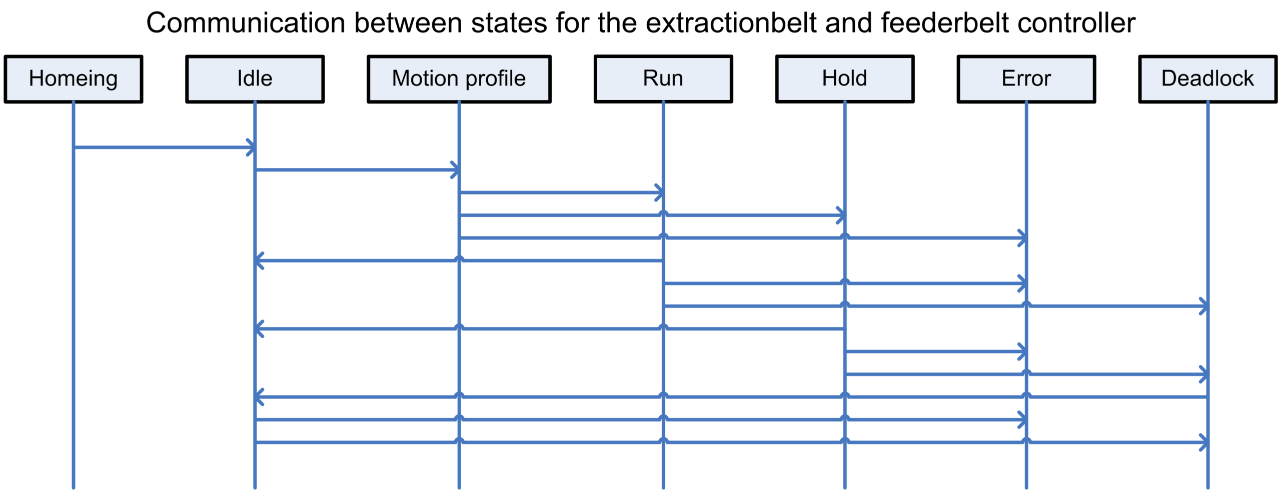
Communication between states for the extractionbelt and feederbelt controller
Homeing
Idle
Motion profile
Run
Hold
Error
Deadlock

Figure 4.5: Communication between states for the extractionbelt and feederbelt controller

4.3 Algorithm

The algorithm, consists of a motion profile generator and a loop controller that contains the PID algorithm, calculates the set point values and PWM value for each motor. It also contains all specific parameters that are needed to calculate the values for each motor. The algorithm runs in an infinite loop through all 6 motors and checks every time if current selected motor is active. If the current motor is activated the set point values and PWM value are calculated and the algorithm continues to the next motor.

The Production Cell works with a sample time of 1 ms, this means that the PWM value for each motor is updated every ms. The result of this sample time is that the algorithm loop has to produce results for all 6 motors within 1 ms. As can be seen from the results of section 3.6, the total calculation time to calculate a PWM value for each motor is 41.54 μs, which results in a lot of free time within this 1 ms. To let the algorithm loop each millisecond a timer function is made.

This timer function has a problem to overcome. Every loop through all 6 motors, a different number of motors can be active. This results in a variable calculation time for each loop. To solve this problem the timer function needs to run in parallel with the algorithm loop function.

The timer function starts with a maximum number that contains the number of clock cycles in 1 ms. This number will be decreased by 1 each clock cycle, when the number reaches zero a go signal to the algorithm loop function will be given. After all values are calculated the algorithm loop is blocked by the go signal. When the go signal from the wait function is received the algorithm loop carries on. The timer function will restart again for the next millisecond.

The algorithm structure shown in figure 4.6 is made in a way that extra motors can be added with not much extra work and FPGA-resource usage. This can be done by adding the acceleration signal to the motion profile part and the specific parameters for the new motor to the PID algorithm part. After this the parameter that contains the number of motors in the loop controller part needs to be added by 1 and the motor is operational.

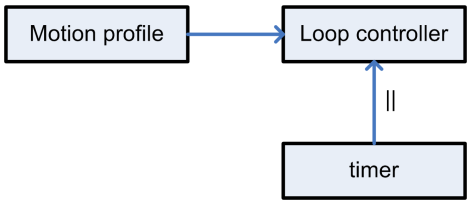

Figure 4.6: Algorithm structure

Table 4.1 is based on the floating-point 32-bit loop controller with motion profile calculation during runtime and shows what will happen to the FPGA-resource usage and calculation time when extra motors are added. As can be seen adding an extra motor to the system cost about 600/3 = 200 LUTs and 21/3 = 7 μs extra calculation time. Code generation of the algorithm with the use of 20-sim and gCSP can be done by replacing operators with functions that are available in the self made floating-point mathematics library.

	LUT	**FF**	**Mem**	**Other**	**ALUs**	**LUT %**	**Mem %**	**C**	**C %**
3 motors	5719	4105	0	1963	4	21,48%	0,00%	14,1 μs	0.01%
6 motors	6385	4531	0	1956	4	23,98%	0,00%	41,54 μs	4.15%
9 motors	7006	6361	0	1930	4	26,31%	0,00%	62,29 μs	6.23%
12 motors	7688	7148	0	1925	4	28,88%	0,00%	83,04 μs	8.3%

Table 4.1: Loop controller FPGA-resource usage and timing with different number of motors (C = Calculation time)

4.4 Safety Layer

The safety layer connects all blocks from the system together. Each signal and value travels through the safety layer, while passing the safety layer the signals will be monitored. This way the safety layer can act when something goes wrong in the system. For example, when a metal block gets stuck inside the molding area. The feeder keeps pushing the block in the molding area while it is stuck. This can cause damage to the molding area and feeder. To prevent this from happening the safety layer can monitor the PWM that goes to the feeder motor and the encoder value that comes from the feeder motor. When the PWM value keeps getting higher while the encoder value of the feeder stays the same, the safety layer knows that something is wrong and can act by stopping the feeder motor and send a signal to all state machines to enter the error state.

The deadlock detection is implemented in a similar way. The safety layer knows all sensor values and can, therefore, determine if the system is in a deadlock or not. When the condition of all sensors triggered is reached it can send the appropriate error signal to all state machine blocks.

4.5 Testing and Results

The new Production Cell structure is tested on the real setup to validate if everything works. This test showed that all motors are running within their operating boundaries and the safety layer reacted on a deadlock and stopped all motors when the feeder got stuck.

The results of the new Production Cell implementation compared to the old integer implementation are shown in table 4.2. The integer version is based on parallel PCU execution and the floating-point version is based on sequential PCU execution. The new implementation uses about 16 % more FPGA-resources, looses the de-centralized structure and is about 10 times slower in calculation time than the integer implementation. However, the new implementation improves the calculation precision range of the system, has an imporoved safety layer and minimizes the design effort.

	LUT	**FF**	**Mem**	**ALUs**	**LUT %**	**Mem %**
Maximum values	26624		589824	32		
Integer Production cell	8444	3272	2816	0	31,72 **%**	0,48 **%**
Floating-point Production Cell	12624	7945	2704	4	47,42 **%**	0,46 **%**

Table 4.2: Production Cell implementation results with PCI debug capability

5 Conclusions and Recommendations

5.1 Conclusions

This project proved that it is possible to control the Production Cell with an FPGA hosted floating-point loop controller that fits in the current design flow while gaining more precision than the integer implementation without handing in too much FPGA-resources.

The floating-point 32-bit loop controller was found to be the best loop controllers from all alternatives. It shows great improvements compared to the integer loop controller on the following subjects: set point generation, precision and design effort.

- The new way of generating set points for the loop controller resulted in less FPGA-resource usage and more accurate set points.
- The floating-point 32-bit loop controller uses float as data type which improves the calculation precision and can reuse the existing loop controller models which leads to less design effort.
- The code generation for the floating-point loop controller from 20-sim and gCSP to FPGA code is possible, which minimizes the design effort.

The fixed-point loop controller still requires to change the current loop controller models, therefore, it did not offer more functionality over the integer version other than more precision.

The new Production Cell framework has an improved safety layer compared to the integer version. It can detect deadlocks and can react on errors that occur in the physical system. The only downside of the new framework is that it is not distributed. However, during this project a parallel version of the floating-point 32-bit loop controller has also been made. This version can be used with the Production Cell framework van Zuijlen (2008) made. This way also a distributed floating-point 32-bit loop controller can be used when a bigger FPGA is available to control the Production Cell with.

5.2 Recommendations

During the investigation on alternative loop controllers the PowerPC and Microblaze option proved to be promising. The ideal platform to test the PowerPC and Microblaze alternatives on is the ML510 FPGA due to support for both PowerPC and Microblaze. The ML510 is currently being investigated by Roderick Colenbrander. When this project is finished the ML510 should be operational to control the Production Cell with the use of a PowerPC or Microblaze. With the ML510 it should also be possible to combine the PowerPC and Microblaze functions resulting in a control system where each part of the Production Cell model runs on a platform it will fit the best on.

The future of Handel-C is unknown. Therefore, it is recommended to look for alternative Hardware Description languages that have the same capabilities as Handel-C. Another solution for this problem is to go back to using VHDL for programming an FPGA. Programming will take more time but the language is lively and has a wide spread support compared to Handel-C.

For the integration of the floating-point Handel-C structure into 20-sim and gCSP regarding the code generation, a converter needs to be made. During this project all values are manually converted with the use of the tool made by Schmidt (2009). It is desirable to integrate the functionality of this tool into 20-sim and gCSP, saving a lot of manual labor.

When using non symmetrical motion profiles the set points need to be stored in memory. With the Spartan 3 FPGA only 15521 set points can be stored. For the Production Cell that was insufficient, hence, A better way of storing set points into the memory is desirable.

The Production Cell uses an IR proximity sensor on the extraction robot side. This sensor proved to be a weak link in the system. On a regular basis the sensor did not detect if a metal block was inside the extraction area. This was causing the extraction arm to grab a metal block out of the molder while a metal block was still in the extraction area causing a collision of the blocks. This problem can be solved by using 2 opto-switches placed in cross link over the extraction belt. Figure 5.1 shows the proposed sensor setup with their object area range and the old sensor with its sensor range. the 2 opto-switches are numbered 1 and 2. The thick black lined square in the center of the switches is the area the switches can see objects in. The 2 thin lines from the current sensor show the current area objects can be detected in. As can be seen, with the new sensors the block in figure 5.1 can be detected while with the current sensor it cannot be detected.

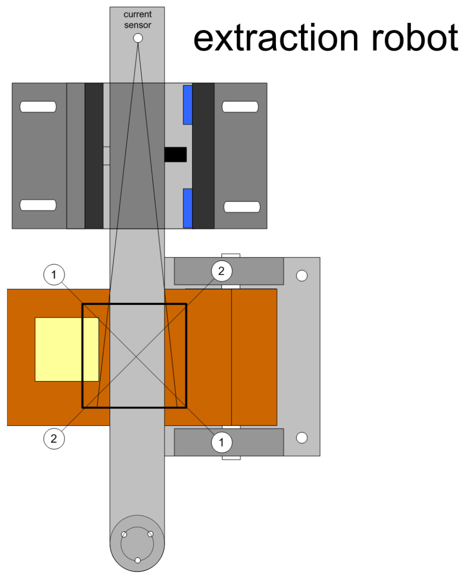

Figure 5.1: Extraction robot with proposed opto-switch sensors

The Production Cell misses a sensor next to the molder door that can detect if there is a metal block inside the molding area. For safety reasons it is desirable to have this sensor. Currently, administration is needed to keep track of metal blocks that are inside the molding area. With this new sensor this administration is not needed saving FPGA-resources.

A Floating-point Converter

This Appendix gives an overview of the test tools that were used during this project.

In order to get information from the test FPGA (RC10) a communication tool had to be made to connect the RC10 with a PC. Celoxica provides a communication library for the RC10 for controlling the USB port. With the use of this library a small program was made to connect and collect data from the FPGA.

The program can flash an image to the FPGA and then waits for data from the FPGA. All data received from the FPGA will be logged into a text file. The program is finished when the FPGA is finished with its task or a time out is triggered. The generated text file with data can be imported into Matlab and the data can be analyzed.

The data coming from the FPGA is formatted in the Handel-C floating-point format which looks like 0, 125, 2013265. This data represents the decimal number 0.31. In order to work with the data coming from the FPGA this structure needs to be converted to decimal numbers.

A new tool was made to convert the Handel-C floating-point structure to decimal numbers. The tool can do two things. The first thing is convert a single floating-point structure to a decimal number or binary code. The second thing is to convert a text file containing a single floating-point structure on each line to a new text file with the corresponding floating-point and binary decimal numbers.

The following code shows an example result of the tool.

```
convert.exe 0 125 2013265
0 125 2013265 converted to binary/decimal is:
00111110100111101011100001010001
0.31
```

B Demo Setup

B.1 Demo Manual

Prerequisites

Before running the demo check if the following things are available and ready to be used:

- PC CE182 (Linux 2.6) with Mesa Anything I/O 5i22 PCI card
- Handel-C bit file (float.bit)
- The Production Cell setup with cables connected to CE182 (so the CORRECT Anything I/O FPGA board) and not to the PC/104
- The Production Cell should not contain any metal blocks.

Running the Demo

1. Start up CE182 and log on (account: demo, password: CE$demo#)
2. Go to the directory /usr/local/ProductionCell
3. Start the demo with the command ./run.sh (Administrator rights are needed to run the demo!!) On CE182, the executable has admin rights for all users (sticky bit))
4. The Production Cell Commander screen will show up
5. Go to the Settings/Override tab and click on connect
6. Go back to the Command tab and click on Start
7. The Production Cell will start the homing procedure
8. Wait 15 seconds until the homing is finished
9. Place metal blocks in the middle of the extraction belt with at least 1 cm width between them.
10. Triggering the extraction belt end sensor (next to the white block platform under the rotation robot) to start the extraction belt.
11. To stop the demo click on the Stop button.
12. Close the Production Cell Commander screen and press enter to erase the FPGA.
13. Turn off CE182 and turn off the Production Cell setup

B.2 Demo Setups

Table B.1 shows all demo setups that are available for the Production Cell. The first 3 are all software based solutions and run with the use of an PC104. The gCSP version also requires the and the Stork Command and Display Unit (SCDU). Demo 4 and 5 are hardware based and run on an FPGA platform.

		Type	**Platform**
1	gCSP	Software	PC104
2	POOSL	Software	PC104
3	Ptolemy	Software	PC104
4	Handel-C integer	Hardware	FPGA
5	Handel-C floating-point	Hardware	FPGA

Table B.1: Production Cell demo setups

C Compiling Handel-C Projects

1 Start up the chosen Handel-C project (*.dkw file)
2 A project can be build for different configurations (Debug, VHDL, EDIF and RC10)

   Debug is for simulation within the DK design suite

   VHDL gives VHDL code of the project

   EDIF produces a bitfile for the Production Cell system

   RC10 produces a bitfile that can be used in the RC10

3 In the Build menu select Set Active Configuration and choose the configuration that is needed
4 By pressing F7 the project will build

D Source Code

The source code of the Floating-point based Production Cell system can be found on the following location:

https://cewiki.ewi.utwente.nl/svn/PRODCELL/trunk/HandelC/Floating Point

D.1 Used Software Versions

20-sim : 4.0.1.7

DK Design suite : DK5.0

Xilinx ISE : 10.1i

Visual studio : .NET 2003 & 6.0 (dll)

Linux : Debian Etch

Bibliography

Arcturus Networks (2009).
`http://www.uclinux.org/`

Barry, R. (2009).
`http://www.freertos.org/`

van den Berg, L. (2006), Design of a Production Cell Setup, MSc Thesis 016CE2006, University of Twente.

Broenink, J. and G. Hilderink (2001), A structured approach to embedded control systems implementation, in *2001 IEEE International Conference on Control Applications*, M. W.Spong, D. Repperger and J. M. I. Zannatha (Eds.), México City, México, pp. 761–766, ISBN 0-7803-6735-9.

Celoxica (2009).
`http://www.celoxica.com/`

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

Groothuis, M. (2009), ViewCorrect.
`http://www.ce.utwente.nl/viewcorrect/`

Groothuis, M., J. van Zuijlen and J. Broenink (2008), FPGA based Control of a Production Cell System, in *Communicating Process Architectures*, IOS press, Amsterdam, NL, pp. 135–148, ISBN 978-1-58603-907-3.

Hoare, C. (1985), *Communicating sequential Processes‘*, Prentice-Hall, ISBN 0131532715.

Huang, J., J. Voeten, M. Groothuis, J. Broenink and H. Corporaal (2007), A model-driven approach for mechatronic systems, in *Seventh International Conference on Application of Concurrency to System Design*, Bratislava, Slovakia, pp. 127–136, ISBN 0-7695-2902-X.

IEEE 754 (2008), IEEE Standard for Floating-Point Arithmetic.
`http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4610935`

Jovanovic, D. S., B. Orlic, G. Liet and J. Broenink (2004), gCSP: A Graphical Tool for Designing CSP systems, in *Communicating Process Architectures*, IOS press, Oxford, UK, pp. 233–251, ISBN 1586034588.

Kahan, W. (1997), Lecture Notes on the Status of IEEE 754, Paper, University of California.
`http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF`

Kernighan, B. and D. Ritchie (1988), *The C programming language*, Prentice Hall, second edition, ISBN 0131103628.

LynuxWorks Inc (2009).
`http://www.lynuxworks.com/`

Maljaars, P. (2006), Control of the Production Cell Setup, MSc Thesis 039CE2006, University of Twente.

occam (1984), *occam Programming Manual*, Prentice-Hall, ISBN 0136292968.

Orlic, B. (2007), *SystemCSP: A graphical language for designing concurrent component-based embedded control systems*, Control Engineering, University of Twente, Enschede, ISBN 978-90-365-2573-2.

QNX Software Systems (2009).
`http://www.qnx.com`

Schmidt, H. (2009).
`http://www.h-schmidt.net/FloatApplet/IEEE754.html`

Silberschatz, A., P. B. Galvin and G. Gagne (2005), Operting System Concepts, John Wiley & Sons, Inc., Hoboken, NJ, pp. 8–10, ISBN 0471694665.

Wind River Systems (2009).
`http://www.windriver.com`

Xilinx (2008).
`http://www.xilinx.com`

van Zuijlen, J. (2007), Feasibility study on Handel C for Embedded Control, Pre-doctoral assignment 014CE2007, University of Twente.

van Zuijlen, J. (2008), FPGA-based control of the production cell using Handel-C, MSc Thesis 008CE2008, University of Twente.