Communicating Process Architectures 2008
P.H. Welch et al. (Eds.)
IOS Press, 2008

FPGA based Control of a Production Cell System

Marcel A. GROOTHUIS, Jasper J.P. VAN ZUIJLEN and Jan F. BROENINK

Control Engineering, Faculty EEMCS, University of Twente,
P.O. Box 217 7500 AE Enschede, The Netherlands.

{M.A.Groothuis,J.F.Broenink}@utwente.nl, J.J.P.vanzuijlen@alumnus.utwente.nl

Abstract. Most motion control systems for mechatronic systems are implemented on digital computers. In this paper we present an FPGA based solution implemented on a low cost Xilinx Spartan III FPGA. A Production Cell setup with multiple parallel operating units is chosen as a test case. The embedded control software for this system is designed in gCSP using a reusable layered CSP based software structure. gCSP is extended with automatic Handel-C code generation for configuring the FPGA. Many motion control systems use floating point calculations for the loop controllers. Low cost general purpose FPGAs do not implement hardware-based floating point units. The loop controllers for this system are converted from floating point to integer based calculations using a stepwise refinement approach. The result is a complete FPGA based motion control system with better performance figures than previous CPU based implementations.

Keywords. embedded systems, CSP, FPGA, Handel-C, gCSP, 20-sim, motion control, PID, code generation.

Introduction

Nowadays, most motion controllers are implemented on programmable logic controllers (PLCs) or PCs. Typical features of motion controllers are the hard real-time timing requirements (loop frequencies of up to 10 kHz). Running multiple controllers in parallel on a single PC can result in missing deadlines when the system load is becoming too high. This paper describes the results of a feasibility study on using a Xilinx Spartan III 3s1500 FPGA for motion control together with a CSP based software framework.

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. Having a general purpose FPGA as motion control platform compared to CPU based implementations has several advantages:

- Parallel execution: no Von Neumann bottleneck and no performance degradation under high system load due to large scale parallelism;
- Implementation flexibility: from simple glue-logic to soft-core CPUs;
- Timing: FPGAs can give the exact timing necessary for motion controllers;
- High speed: directly implementing the motion controller algorithms in hardware allows for high speed calculations and fast response times. Although not directly required for the chosen system this can, for example, be beneficial for *hardware-in-the-loop* simulation systems. Typical PC based solutions can reach up to 20-40 kHz sampling frequencies, while FPGA based solutions can reach multi-MHz sampling frequencies.

A main disadvantage is that a general purpose FPGA is not natively capable of doing floating point calculations, which are commonly used in motion control systems. For more information on FPGAs and their internal structure, see [1].

One of our industrial partners in embedded control systems is moving from their standardised CPU + FPGA platform towards an FPGA-only platform. A soft-core CPU implemented on the FPGA is used to execute the motion controllers. This approach however still suffers from the Von Neumann bottleneck and the implementation of a soft-core CPU requires a large FPGA.

The target for this feasibility study is a mock-up of a Production Cell system (see figure 1) based on an industrial plastic molding machine. This system consists of 6 moving robots that are each controlled by a motion controller. The previous implementation of the system software was running on an embedded PC. The motion controllers in this implementation suffer from performance degradation when the system is under high load (when all moving robots are active at the same time). The Production Cell system already contains an FPGA. It is currently only used as an I/O board (PWM generators, quadrature encoder interfaces and digital I/O), to interface the embedded PC with the hardware.

The problems with the software implementation, the possible benefits of using an FPGA and the move towards FPGA-only platforms resulted in this feasibility study in which we wanted to implement a motion control system inside an FPGA without using a soft-core CPU.

We have used a model based design approach to realize the FPGA based motion control implementation for this setup. The tools 20-sim [2] and gCSP [3] are used to design the loop-controllers and the embedded control software. The CSP process algebra and the Handel-C hardware description language [4] are used in combination with code-generation from 20-sim and gCSP for the design and implementation of the embedded control software.

Section 1 gives more background information on the production cell setup, our previous experiments, motion control and our model based design method. Section 2 describes the designed software framework and section 3 describes the consequences for the design of the loop controllers when running them on an FPGA. This paper concludes with the results (section 4) and conclusions of this feasibility study and future work.

1. Background

1.1. Production Cell

An industrial Production Cell system is a production line system consisting of a series of actors that are coordinated to fulfill together a production step in a factory. The production cell system that is used for this feasibility study is a mock-up designed to resemble a plastics molding machine that creates buckets from plastic substrate. The system consists of several devices that operate in parallel [5]. Its purpose is to serve as a demonstrator for CSP based software, distributed control and to prototype embedded software architectures. Figure 1 shows an overview of the setup.

The setup is a circular system that consists of 6 robots that operate simultaneously and need to synchronize to pass along metal blocks. In this paper each of these robots is called a Production Cell Unit, or PCU. Each PCU is named after its function in the system (see also figure 1). The operation sequence begins by inserting a metal block (real system: plastic substrate) at 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 (real system: creating a bucket from the plastic substrate) takes place. The feeder retracts and the molder door opens. The *extraction robot* can now extract the block (real system: bucket) from the molder. The block is placed on the *extraction belt* which transports it to the *rotation robot*. The rotation robot picks up the block from the extraction belt and puts

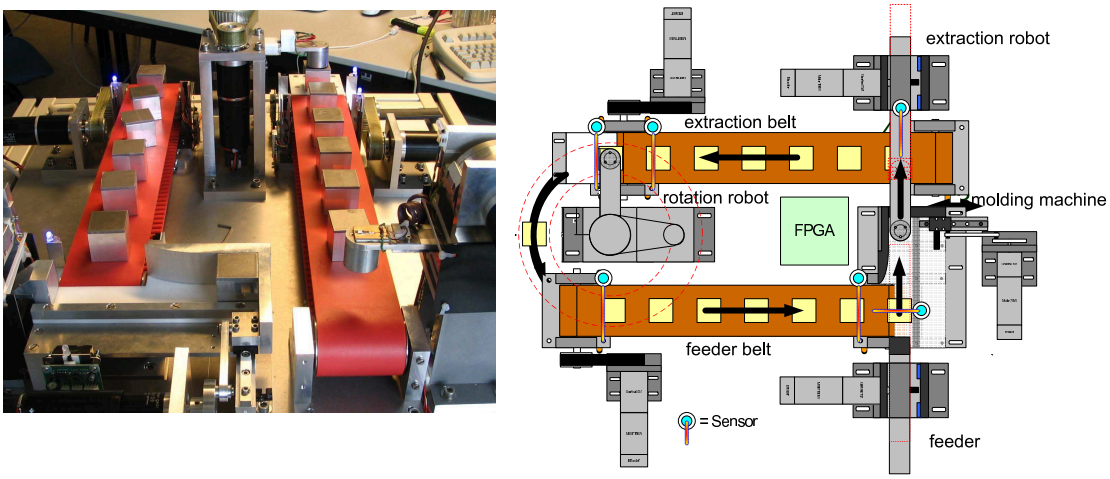

Figure 1. The Production Cell setup

it again on the feeder belt to get a loop in this demonstration setup. This loop can also result in a nice (for teaching purposes) deadlock when 8 or more blocks are in the system. This deadlock occurs when all sensor positions are occupied with blocks, resulting in the situation that all robots are waiting for a free position (at the next sensor), in order to move their block forward.

The belts allow for multiple blocks to be buffered so that every PCU can be provided with a block at all times, allowing all PCUs to operate simultaneously. The blocks are picked up using electromagnets mounted on the extraction robot and the rotation robot. Infrared detectors (sensors in figure 1) are used for detection of the blocks in the system. They are positioned before and after each PCU.

1.2. Previous Experiments

Several other software based solutions have been made in the past to control the Production Cell setup. The first implementation [6] is made using gCSP [7] in combination with our CTC++ library [8] and RTAI (real-time) Linux. 20-sim [2] is used to model the system dynamics and to derive the control laws needed for the movements in the system. Its purpose was to evaluate gCSP/CTC++ for controlling a complex mechatronic setup. This software implementation operates correctly when the system is not overloaded with too many blocks. When all 6 PCUs are active and many sensors are triggered at the same time, the CPU load reaches 100%, resulting in a serious degradation of system performance, unsafe operation, and sometimes even in a completely malfunctioning system. Another implementation [9] is made using the Parallel Object Oriented Specification Language (POOSL [10], based on Milner's CCS [11]). The main focus for this implementation was on the combination of discrete event and continuous time software, the design method and predictable code generation. The properties (e.g. timing, order of execution) of the software model should be preserved during the transformation from model to code. This implementation also could not guarantee meeting deadlines for control loops under high system load. Furthermore, neither implementation incorporates safety features in its design.

1.3. gCSP

gCSP is our graphical CSP tool [7] based on the graphical notation for CSP proposed by Hilderink [12]. gCSP diagrams contain information about compositional relationships (SEQ, PAR, PRI-PAR, ALT and PRI-ALT) and communication relationships (rendezvous channels).

An example of a gCSP diagram with channels, processes and SEQ and PAR compositions is given in figure 3. From these diagrams gCSP is able to generate CSPm code (for deadlock and livelock checking with FDR2/ProBE), occam code and CTC++ code [8]. Recent additions to gCSP are the Handel-C code generation feature (see section 4) and animation/simulation facilities [13].

1.4. Handel-C

Handel-C [4] is an ANSI C based hardware description language born out of the idea to create a way to map occam programs onto an FPGA. Handel-C uses a subset of ANSI C, extended with CSP concepts like channels and constructs. Its built-in support for massive parallelism and the timing semantics (single clock tick assignments) are the strongest features of Handel-C. The close resemblance with the C programming language makes it a suitable target for tools with C based code generation facilities. This was one of the reasons that Rem et al. [14] used Handel-C as a code generation language together with MATLAB/Simulink to design an FPGA based motion controller. While Simulink can be used to generate FPGA optimized motion controller code by using Handel-C templates for each library block, it does not support the design of a software framework with multiple parallel processes containing these motion controllers (targeted for FPGA usage). gCSP is more suited for this purpose.

1.5. Motion Control

Typical motion control systems consist of motion profiles (the trajectory to follow) and loop controllers. Their purpose is to control precisely the position, velocity and acceleration of rotational or translational moving devices, resulting in a smooth movement. The control laws for the loop controllers require a periodic time schedule in which jitter and latency are undesired. Hard real-time behaviour is required for the software implementation, to assure predictable timing behaviour with low latency and jitter. Missing deadlines may result in a catastrophic system failure. The embedded control software of a motion control system often contains a layered structure [15] as shown in Fig. 2 .

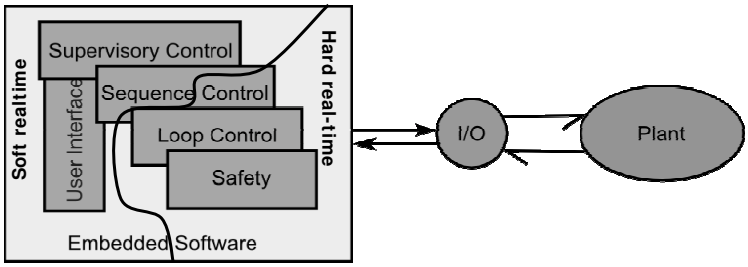

Figure 2. Embedded control system software structure

The typical software layers in motion control systems are:

- Man-machine/user interface;
- Supervisory control;
- Sequence control;
- Loop control;
- Data analysis;
- Measurements and actuation.

Besides a functional division in layers from a control engineering point of view, a division can also be made between hard real-time and soft real-time behaviour: the closer the software layer is to the machine or plant, the more strict the timing must be. Hence, the su-

pervisory control and parts of the sequence control are soft real-time, and mostly run at lower sampling frequencies. In the case of the Production Cell, at least loop controllers (including motion profiles) and sequence controllers (to determine the order of actions) are needed.

1.6. Design Method

To structure the design process for these kind of systems, we use the following design approach:

- Abstraction;
- Top-down design;
- Model-based design;
- Stepwise refinement, local and predictable, aspect oriented

For the system software this means that we start with a top-level abstraction of the system that is refined towards the final implementation. During these stepwise refinements we focus on different aspects (e.g. concurrency, interactions between models of computation, timing, predictable code generation) of the system. To design the loop controller, we follow a similar stepwise refinement approach. The first step is *physical system modelling*: model and understand the plant dynamics. The second step is *control law design*: design a proper control law for the required plant movements. The third step is the *embedded control system implementation* phase in which relevant details about the target are incorporated in the model. These include the non-idealness of the interfaces with the outside world (sampling, discretization, signal delays, scaling), target details (CPU, FPGA). This step ends with code generation and integration of the loop controllers into the systems embedded software. Verifications by simulation are used after the first three steps. Validation and testing are done on the last step *realization*.

In the following two sections, the above design method is applied on the production cell FPGA design.

2. Structure and Communication

This section describes the design and implementation of the CSP based structural and communication (S&C) framework in which the loop controllers are embedded. First, requirements are formulated, after which the design is constructed in a top-down way.

2.1. Requirements

To focus the experiments and tests, the following requirements are formulated:

- Decentralised design to allow distribution across multiple FPGAs (or CPUs) if needed. This means that each PCU must be able to operate independently and that a central supervisory controller is missing.
- CSP based. Exploit parallelism. The setup consists of parallel operating robots, so the natural parallelism of the set up will be exploited.
- Generic. It should be usable for both a software and hardware implementation and for other mechatronic setups.
- Layered structure. This setup should be representative for industrial-sized machine control. Support for hierarchy using the layered structure is inevitable.
- Safety software distinguished from the normal operation. Handling faults can best be separated from the normal operation. This better structures the software, so that parts can be tested individually. Furthermore, design patterns about fault handling strategies can be used here.

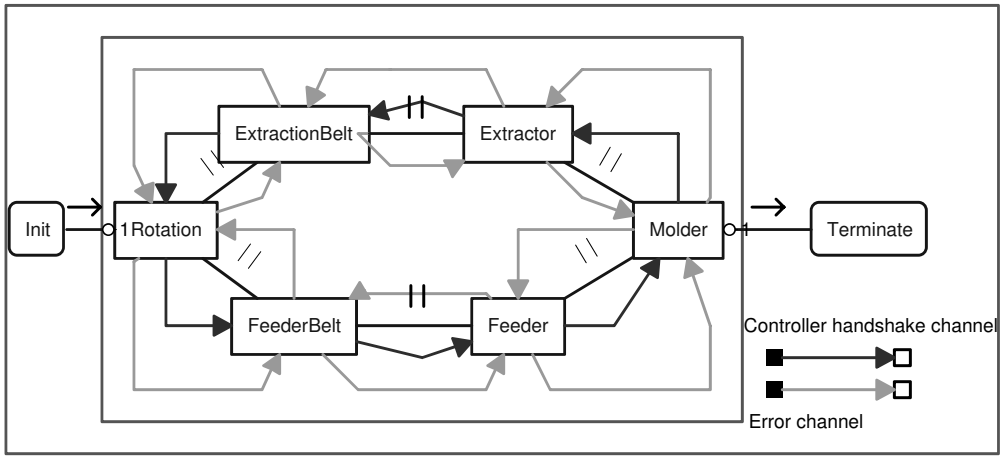

Figure 3. Top-level gCSP diagram

2.2. Top Level Design

We have chosen to implement the 'software' in a layered structure, taking into account the above requirements. The resulting top-level design is shown in figure 3. It shows an abstract view of the Production Cell system with PCUs implemented as parallel running CSP processes. Each PCU is connected to its neighbours using rendezvous channels. No central supervisory process exists and the PCUs are designed such that they are self sustaining. Since the production cell setup has a fixed direction for the blocks (*feeder belt* > *feeder* > *molder door* > *extractor* > *extraction belt* > *rotation*), normal communication is only necessary with the next PCU. The communication is a handshake (CSP rendezvous) for transporting a block. This normal communication will be called the *normal-flow* of the system. When a failure occurs, communication with both neighbours is required. For instance, when the feeder is stuck, not only should the molder door be opened, but also the feeder belt should be stopped in order to stop the flow of blocks. The next sections describe the design of the PCUs in more detail.

2.3. Production Cell Unit Design

A PCU is designed such that most of its operation is independent of the other PCUs. Each PCU can be seen as an independent motion control system. Communication with its neighbours is only needed for delivering a block to the next PCU, or in case of local failures that need to be communicated to both neighbours. Based on the layered structure described in section 1.5 and other implementations [16,15,17] a generic CSP model is made for all 6 PCUs. Figure 4 shows this PCU model, containing three parallel running processes. The controller process implements the motion controller intelligence via a sequence controller and a loop controller. Section 2.5 describes the controller in more detail. The command process implements the Man-machine interface for controlling a PCU from a PC (e.g. to request status info or to command the controller). The safety process contains data analysis intelligence to detect failures and unsafe commands. All communication between the setup, the command interface and the controller is inspected for exceptional conditions. The safety layer will be explained in more detail in the next section. The low-level hardware process contains the measurement and actuation part. Quadrature encoder interfaces for position measurement of the motors, digital I/O for the magnets and the block sensors and PWM generators to steer the DC motors are implemented here.

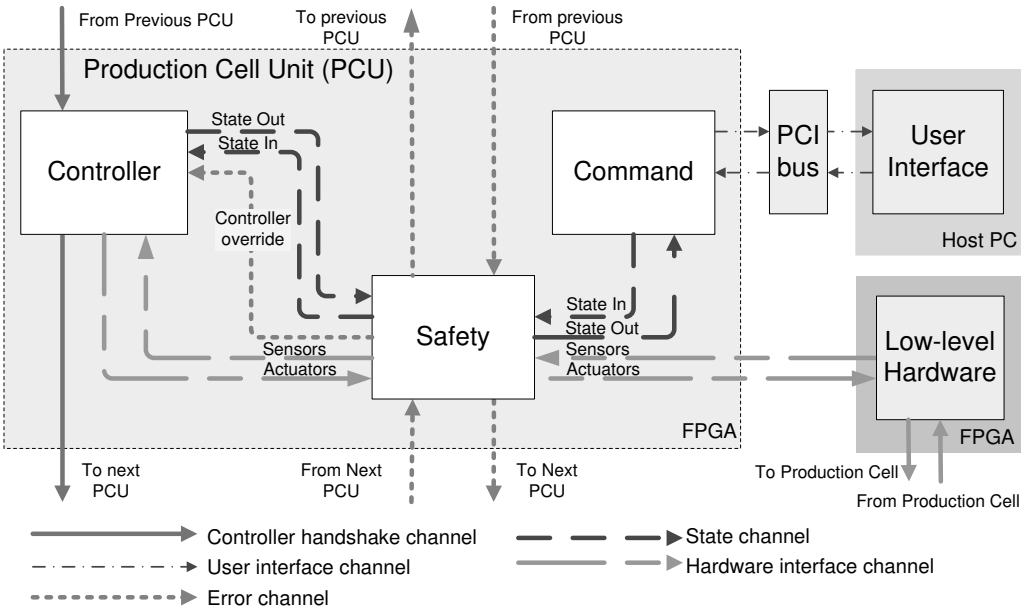

Figure 4. Production Cell Unit – Model

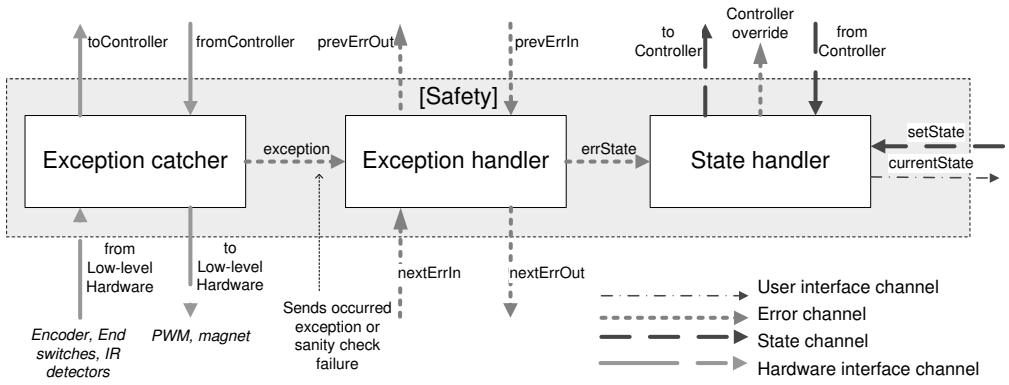

Figure 5. Production Cell Unit – Safety

2.4. Safety

The safety process implements a safety layer following the general architecture of protection systems [18] and the work of Wijbrans [19]. The safety consists of three stages: the *exception catcher*, the *exception handler* and the *state handler* (see figure 5). The *exception catcher* process catches exceptions (hardware to *controller* errors) as well as sanity check failures (*controller* to hardware errors). It sends an error message to the *exception handler*, which converts the error message into three state change messages:

- Its own (safe) controller state via the *errState* channel;
- A safe controller state for the previous PCU in the chain;
- A safe controller state for the next PCU in the chain.

The *state handler* process controls the states in a PCU and is the link between the *normal flow* and the error flow. Here the decision is made what state is being sent to the controller process (figure 4). It receives state information from the *Exception handler* process, the *Controller* process and the *User interface*.

The highest priority channel is the *errState* state channel from the exception handler. This channel transports the ‘safe’ state from the exception handler to the state handler when a failure has occurred. Once this channel is activated, this state will always be sent to the *Con-*

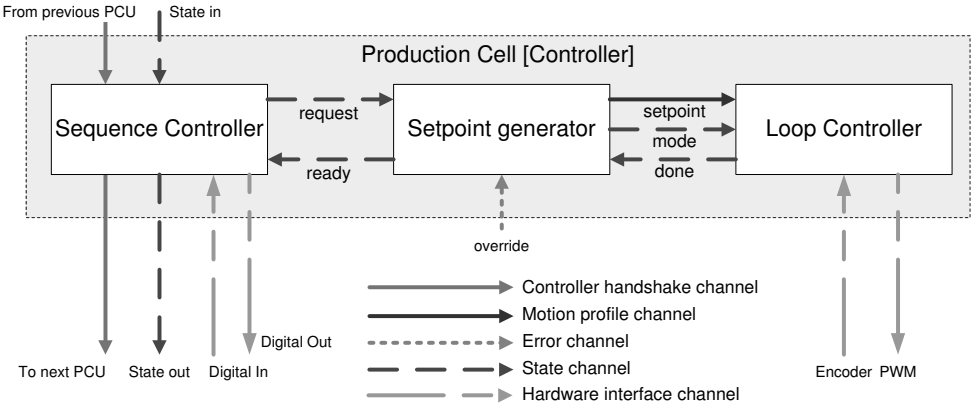

Figure 6. Production Cell Unit – Controller operation

troller (figure 4). The *override* channel is activated as well in order to keep the neighbouring PCU in its state until the error has been resolved.

2.5. Controller

Figure 6 shows the internals of the controller process. The controller process consists of a *sequence controller*, a *setpoint generator* and a *loop controller*. The sequence controller acts on the block sensor inputs and the rendezvous messages from the previous PCU. It determines which movement the PCU should make. It controls the *setpoint generator* that contains setpoints for stationary positions and it is able to generate motion profiles for movements between these stationary positions. The *loop controller* receives setpoints from the generator. Dependent on the mode set by the *setpoint generator* the loop controller is able to:

- Run a homing profile;
- Execute a regulator control algorithm (to maintain a stationary position);
- Execute a servo control algorithm (to track motion profiles).

The homing profile mode is needed to initialize the quadrature encoder position sensors on the motor at start-up. The design of the loop controller algorithm is explained in section 3.

2.6. Communication Sequence

The process framework is now almost complete. The framework is now capable of communicating with other PCUs and it can safely control a single PCU. This section briefly describes the interactions between the PCUs: the startup phase, handshaking and communication.

2.6.1. Normal Flow

When the hardware setup is turned on, all PCUs execute their homing action for sensor initialization and as a startup test. After the homing phase, the system is idle until a block is introduced and triggers a sensor. Figure 7 shows an example of the communication between the PCUs when a single block makes one round starting at the *feeder belt*.

2.6.2. Error Flow

In case a failure occurs, for example a block is stuck at the *feeder*, the local *exception catcher* and *exception handler* will put the *feeder* in a safe state and communicate to the neighbours (the feeder belt and the molder door) that it has a problem. The *molder door* will open and the *feeder belt* will stop supplying new blocks.

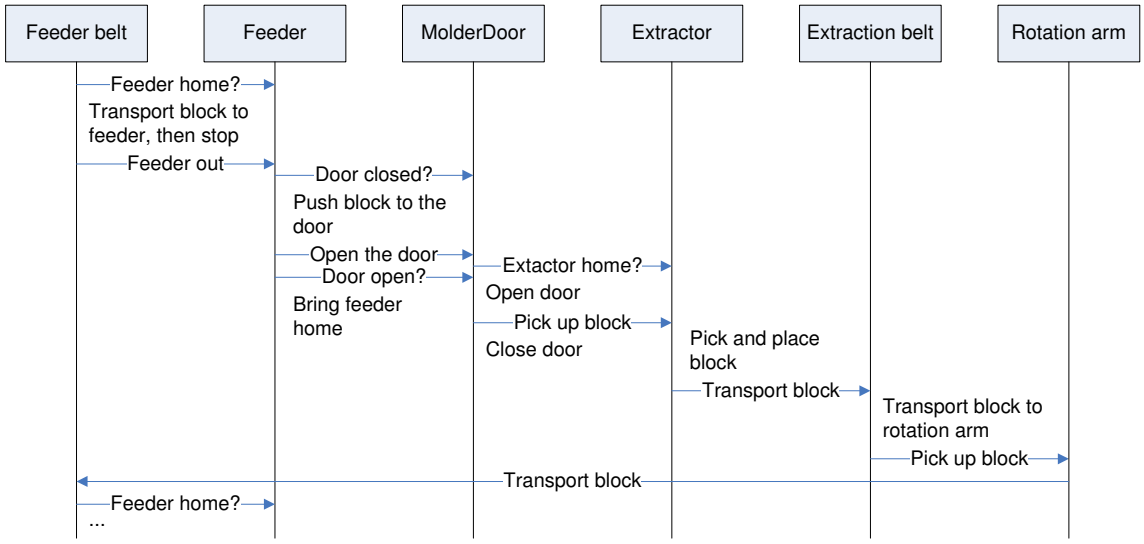

Figure 7. Production Cell – Normal operation sequence diagram

3. Loop Controller Design

An important part of this feasibility study is the implementation of loop-controllers in an FPGA. The control laws for these loop-controller are designed via step-wise refinement in 20-sim using a model of the plant behaviour. The design of these loop-controllers was originally done for software implementations. The resulting discrete-time PID[1] loop controllers and motion profiles used floating point calculations. The PID controller [20] is based on a computer algorithm that relies on the floating point data type (see listing 1 for the algorithm). Its purpose is to minimize the *error* between the current position and the desired position. This error is typically a small value (for the PCUs at most 0.0004m) so some calculation accuracy is needed here. The chosen FPGA has no on-board floating point unit (FPU), so another solution is needed here.

3.1. Floating Point to Integer

Table 1 shows some alternatives to using a floating-point data type.

Table 1. Alternatives to using the floating point data type on an FPGA

	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.	External FPU	High precision; re-use existing controller	Only available on high end FPGAs; expensive
4.	Soft-core CPU+FPU	High precision; re-use existing controller	High logic utilization unless stripped
5.	Soft-core FPU	High precision; re-use existing controller	Scheduling / resource manager required
6.	Integer	Native datatype	Low precision in small ranges; adaptation of the controllers needed

The numerical precision is coupled to the logic cell utilization, resulting in a design trade-off between numerical precision and FPGA utilization [21]. Agility, a provider of em-

[1]Proportional, Integral, Derivative.

```
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;
```

Listing 1. The PID loop-controller algorithm

bedded systems solutions formed from the merger of Catalytic Inc. and Celoxica's ESL business, delivers Handel-C libraries for both floating point and fixed point calculation. A main drawback of the first two options is that the resulting FPGA implementations have very high logic utilization because each calculation gets its own hardware. This is not a viable alternative for the chosen FPGA (a small test with 1 PID controller resulted in a completely filled FPGA for floating point). The third option requires a high-end FPGA with DSP facilities. The fourth option to use a soft-core CPU with a floating point unit (e.g. a Xilinx Microblaze CPU with single precision FPU which costs around 1800 LUTs[2]). The advantage is that we still can use our existing loop controllers (from the previous software version). The drawback is that the design becomes more complicated, due to the combination of Handel-C hardware and soft-core CPU software. Furthermore, we need a scheduler if all 6 PID controllers should run on the same soft-core CPU. The soft-core CPU solution is excessive for just a PID controller. An FPU-only soft-core is a better choice here, but still some scheduling is needed. The last option, integer based calculation, is the most suitable for efficient FPGA usage. However, this requires a redesign of the PID controllers. Despite the disadvantages of switching to an integer based PID controller, we have chosen this solution because the first three options are unfeasible for our FPGA and our goal was to not use a soft-core CPU.

To make the PID algorithm suitable for integer based control, taking into account the needed numerical precision, the following conversions are necessary:

- Integer based parameters;
- Integer based mathematics;
- Proper scaling to reduce the significance of fractional numbers;
- Take into account the quantization effects of neglecting the fractional. numbers

The original controllers used SI-units for I/O and parameters, resulting in many fractional numbers. All signals and parameters are now properly scaled, matching the value ranges of the I/O hardware (PWM, encoder). The conversions mentioned earlier are executed via step-wise refinement in 20-sim using simulations and a side-by-side comparison with the original floating point controllers. The new integer based controllers are validated on the real setup using the CPU based solution, to make sure that the resulting FPGA based integer PID controllers have a similar behaviour compared to the original floating-point version. Some accuracy is lost due to the switch to integer mathematics, resulting in a slightly larger error (0.00046m).

4. Realization and Results

The embedded control software structure for the production cell setup from section 2 was first checked for deadlocks using a separate gCSP model. The top-level structure in figure 3 is extended with an extra block inserter process. An FDR2 test shows indeed a deadlock with 8 blocks or more in the system as described in section 1.1.

For the Handel-C implementation, the gCSP model of figure 3 is refined in a systematic way to a version suitable for automatic code generation. To support automatic code gener-

[2]Look-Up Table, and a measure of FPGA size/utilization, with one LUT for each logic block in the FPGA.

```
void Rotation(chan* eb2ro_err, chan* ro2eb_err, chan* fb2ro_err, chan* ro2fb_err,
              chan* eb2ro, chan* ro2fb)
{
  /* Declarations */
  chan int cnt0_w encoder_in;
  chan int 12 pwm_out;
  chan int 2 endsw_in;
  chan int 1 magnet_out;
  chan int state_w setState;
  chan int state_w currentState;
  chan int state_w saf2ctrl;
  chan int state_w override;
  chan int 12 ctrl2hw;
  chan int state_w ctrl2saf;
  chan int cnt0_w hw2ctrl;
  chan int 1 magnet_saf;

  /* Process Body */
  par {
    LowLevel_hw(&encoder_in, &pwm_out, &endsw_in, &magnet_out);
    seq {
      Init(&encoder_in, &magnet_out, &pwm_out);
      par {
        Command(&setState, &currentState);
        Safety(&eb2ro_err, &saf2ctrl, &ro2eb_err, &override, &encoder_in,
               &fb2ro_err, &pwm_out, &setState, &ro2fb_err, &ctrl2hw,
               &currentState, &ctrl2saf, &hw2ctrl);
        Controller(&saf2ctrl, &override, &eb2ro, &ctrl2hw, &ctrl2saf,
                   &ro2fb, &hw2ctrl, &magnet_saf);
      }
      Terminate(&encoder_in, &magnet_out, &pwm_out);
    }
  }
}
```

Listing 2. Generated Handel-C code for the Rotation PCU

ation, gCSP is extended with Handel-C code generation capabilities. Due to the CSP foundation of gCSP, mapping the gCSP diagrams to Handel-C code was rather straightforward. Because Handel-C does not support the ALT and PRI-PAR constructs (only PRI-ALT and PAR are supported) some drawing restrictions were added. Furthermore, gCSP was extended with the possibilities to add non-standard datatypes to be able to use integer datatypes of a specific width. Listing 2 shows an example of the gCSP generated Handel-C code for the *rotation* PCU. This PCU is implemented in gCSP using the design shown figure 4 (the *Init* and *Terminate* blocks for the hardware are not shown in this figure).

The loop-controllers are implemented using a manually adapted version of the code that 20-sim has generated. Currently, 20-sim generates only ANSI-C floating-point based code. The Handel-C integer PID controller is first tested stand-alone in a one-to-one comparison with an integer PID running on the PC containing our FPGA card.

To be able to see what is happening inside the FPGA and to test the PID controllers and the *Command* process, we have implemented a PCI bus interface process (see also figure 4) to communicate between our development PC and the FPGA. This has proved to be a useful debugging tool during the implementation phase. Currently we are using the PCI debugging interface in cooperation with a Linux GUI program to show the internal status of the PCUs and to manually send commands to the FPGA.

Table 2 shows some characteristics of the realized FPGA implementation to get an idea of the estimated FPGA usage for this system. The total FPGA utilization for the Spartan III 1500 is 43% (measured in slices).

The behaviour of the Production Cell setup is similar to the existing software implementations. Compared to the existing CPU based solutions, the FPGA implementation shows

Table 2. Estimated FPGA usage for the Production Cell Motion Controller

Element	LUTs (amount)	Flipflops (amount)	Memory
PID controllers	13.5% (4038)	0.4% (126)	0.0%
Motion profiles	0.9% (278)	0.2% (72)	0.0%
I/O + PCI	3.6% (1090)	1.6% (471)	2.3%
S&C Framework	10.3% (3089)	8.7% (2616)	0.6%
Available	71.7% (21457)	89.1% (26667)	97.1%

perfect performance results under high system load (many blocks in the system) and all hard real-time constraints are met. The controller calculations are finished long before the deadline. Usage of the Handel-C timing semantics to reach our deadlines is not needed with a deadline of 1 ms (sampling frequency of 1 kHz). The PID algorithm itself requires only 464 ns (maximum frequency 2.1 MHz).

The performance of the FPGA based loop controllers is comparable to the CPU based versions. No visible differences in the PCU movements are observed and the measured position tracking errors remain well within limits. An additional feature of the FPGA solution is the implementation of a safety layer, which was missing in the software solutions.

5. Conclusions and Future work

The result of this feasibility study is a running production cell setup where the embedded control software is completely and successfully implemented in a low-cost Xilinx Spartan III XC3s1500 FPGA, using Handel-C as a hardware description language. The resulting software framework is designed such that it is generic and re-usable in other FPGA based or CPU based motion control applications. An FPGA only motion control solution is feasible, without using a soft-core CPU solution. The switch from CPU based implementations towards an FPGA based solution resulted in a much better performance with respect to the timing and the system load. However, the design process for the loop controllers requires more design iterations to ensure that a switch from floating-point calculations to integer based calculations results in correct behaviour.

The potential for FPGA based motion control systems running multiple parallel controllers is not limited to our production cell system. It is also a suitable alternative for our humanoid (walking) soccer robot that contains 12 controllers and a stereo vision system [22].

Although not needed for this setup, the implemented PID controller can reach frequencies of up to 2.1 MHz, which is impossible to achieve on a PC (maximum 40 kHz). This means that other applications requiring high controller frequencies can benefit from an FPGA based controllers.

While this feasibility study shows the potential of using a low-cost FPGA for complex motion control systems, there is still room for improvement and further investigation.

Table 2 shows that the PID controllers take almost half of the required FPGA cells. We have now implemented 6 dedicated PID controllers. A possible optimization would be to implement one PID process and schedule the calculations. We have enough time left to serialise the calculations. However, this conflicts with our goal of exploiting parallelism within the FPGA.

The process for designing integer based motion controllers should be simplified. 20-sim currently has too little support for assisting in the design of integer based motion controllers. Research on the topic of integer based control systems could potentially result in better design methods. Besides this, it would also be a good idea to evaluate the other implementation possibilities from table 1 (especially the soft-core with FPU option), to compare and explore

these design space choices. In this way we can better advise on what to use for FPGA based motion control systems in which situations.

Integer based control systems need further research from the control engineering point of view. Especially with respect to accuracy and scaling effects. This is not only needed for FPGA based designs but also for microcontroller targets and soft-core CPUs without an FPU.

While the software framework was successfully designed using gCSP and its new Handel-C code generation output, there are opportunities for improvement in order to facilitate the future design of production cell-like systems. The structure and communication framework can be re-used, so having the option of using library blocks or gCSP design templates would speed-up the design process. Furthermore, only a subset of the Handel-C and the gCSP language (GML) is supported by the code-generation module. This should be extended.

References

[1] Clive Maxfield. *The Design Warriors Guide to FPGAs, Devices, Tools, and Flows*. Mentor Graphics Corp., 2004. www.mentor.com.

[2] Controllab Products B.V. 20-sim, 2008. http://www.20sim.com.

[3] Dusko S. Jovanovic, Bojan Orlic, Geert K. Liet, and Jan F. Broenink. gCSP: A Graphical Tool for Designing CSP systems. In Ian East, Jeremy Martin, Peter H. Welch, David Duce, and Mark Green, editors, *Communicating Process Architectures 2004*, pages 233–251. IOS press, Oxford, UK, 2004.

[4] Agility Design Systems. Handel-C, 2008. http://www.agilityds.com.

[5] L.S. van den Berg. Design of a production cell setup. MSc Thesis 016CE2006, University of Twente, 2006.

[6] Pieter Maljaars. Control of the production cell setup. MSc Thesis 039CE2006, University of Twente, 2006.

[7] D.S. Jovanovic. *Designing dependable process-oriented software, a CSP approach*. PhD thesis, University of Twente, Enschede, NL, 2006.

[8] Bojan Orlic and Jan F. Broenink. Redesign of the C++ Communicating Threads library for embedded control systems. In Frank Karelse, editor, *5th PROGRESS Symposium on Embedded Systems*, pages 141–156. STW, Nieuwegein, NL, 2004.

[9] Jinfeng Huang, Jeroen P.M. Voeten, Marcel A. Groothuis, Jan F. Broenink, and Henk Corporaal. A model-driven approach for mechatronic systems. In *IEEE International Conference on Applications of Concurrency to System Design, ACSD2007*, page 10, Bratislava, Slovakia, 2007. IEEE.

[10] Bart D. Theelen, Oana Florescu, M.C.W. Geilen, Jinfeng Huang, J.P.H.A van der Putten, and Jeroen P.M. Voeten. Software / hardware engineering with the parallel object-oriented specification language. In *ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE2007)*, pages 139–148, Nice, France, 2007.

[11] Robin Milner. *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, 1989.

[12] Gerald H. Hilderink. *Managing Complexity of Control Software through Concurrency*. PhD thesis, University of Twente, Netherlands, 2005.

[13] T.T.J. van der Steen. Design of animation and debug facilities for gCSP. MSc Thesis 020CE2008, University of Twente, 2008.

[14] B Rem, A Gopalakrishnan, T.J.H. Geelen, and H.W. Roebbers. Automatic Handel-C generation from MATLAB and simulink for motion control with an FPGA. In Jan F. Broenink, Herman W. Roebbers, Johan P.E. Sunter, Peter H. Welch, and David C. Wood, editors, *Communicating Process Architectures CPA 2005*, pages 43–69. IOS Press, Eindhoven, NL, 2005.

[15] S. Bennett. *Real-Time Computer Control: An Introduction*. Prentice-Hall, London, UK, 1988.

[16] Herman Bruyninckx. Project Orocos. Technical report, Katholieke Universiteit Leuven, 2000.

[17] S. Bennett and D.A. Linkens. *Real-Time Computer Control*. Peter Peregrinus, 1984.

[18] P. A Lee and T. Anderson. *Fault tolerance, principles and practice*. Springer-Verlag, New York, NY, 1990.

[19] K.C.J. Wijbrans. *Twente Hierarchical Embedded Systems Implementation by Simulation (THESIS)*. Universiteit van Twente, 1993.

[20] Karl J. Åström and T. Hagglund. *PID Controllers: Theory, Design and Tuning*. ISA, second edition, 1995.

[21] Michael J. Beauchamp, Scott Hauck, Keith D. Underwood, and K. Scott Hemmert. Embedded floating-point units in FPGAs. In Steven J. E. Wilton and André DeHon, editors, *FPGA*, pages 12–20. ACM, 2006.
[22] Dutch Robotics. 3TU humanoid soccer robot, TUlip, 2008. http://www.dutchrobotics.net.