An FPGA based motion controller

Mehmet Fatih Ozturk
Alper Genc
2011
SUMMARY

This document is the final report for the graduation project of the Embedded Systems Master of Science program which was carried out by Mehmet Fatih Ozturk and Alper Genc in ASML within the last 8 months. This assignment focuses on wafer scanner products that ASML produces. These products are composed of several components. One of the most critical parts of the products is the illumination part. In the illumination part, a new technology called FlexRay is introduced that enables clients of ASML to create laser beams in desired shapes. It is composed of many tiny mirrors. For each mirror there is a controller to control the motion of the mirror. A high sample frequency mirror controller is required to control these mirrors. This assignment aims to define the architecture for an FPGA based motion controller for FlexRay.

The objectives of this study include:

- Investigating the reuse of a technology called the RingBus based Motion Controller within ASML which has already been implemented on an FPGA.
- Defining an architecture that conforms to the architectural reference model called CARM and includes the reusable parts from the RingBus based Motion Controller.
- Verifying that the design satisfies the performance and auxiliary function requirements.

In this assignment, first of all, a preliminary investigation is done. Based on the conclusions of this investigation and the given requirements, the architecture is defined for an FPGA-based motion controller. The design follows a stepwise refinement approach. The motion controller is gradually decomposed towards the FPGA components. In each step of the refinement, the design is verified. Furthermore, a start is made to implement the design to be implemented on an FPGA so the solution can be completed in a future assignment.

MANAGEMENT ADVICE

The project results include the output of the preliminary investigation work, a system-level design and a detailed design document for the FPGA-based motion controller. Through the preliminary investigation work, it can be concluded that it is feasible to integrate some components of the RingBus based Motion Controller architecture to an FPGA based FlexRay motion controller architecture. Together with the system level and detailed design, the following conclusions can be made:

- It is feasible to map the motion controller to an FPGA based platform.
- Two FPGA boards are enough to satisfy the given requirements.
- For higher frequencies (~40x faster), an FPGA based motion controller has advantages compared to the CPU-based motion controller:
  - Rack space decreases remarkably (~15x smaller).
  - Power consumption decreases tremendously (~30x less).
  - Cost decreases impressively (~30x cheaper).

To be more confident about the feasibility of the FPGA-based solution, we strongly recommend as a first step for ASML to complete the implementation and conduct experiments by following the approach we elaborate in Chapter 6.
TABLE OF CONTENTS

1 INTRODUCTION .......................................................................................................................... 8

2 FLEXRAY .................................................................................................................................. 10

3 PROBLEM STATEMENT .............................................................................................................. 11
  3.1 DESIGN CONSTRAINTS ........................................................................................................... 11
  3.2 PERFORMANCE REQUIREMENTS ........................................................................................... 11
  3.3 AUXILIARY FUNCTIONS ......................................................................................................... 12

4 PRELIMINARY INVESTIGATION ................................................................................................. 14
  4.1 SPU & RINGBUS .................................................................................................................... 14
    4.1.1 SPU .................................................................................................................................. 14
    4.1.1.1 SCHEDULING MECHANISM......................................................................................... 16
    4.1.2 RINGBUS ........................................................................................................................ 17
  4.2 APPROACH FOR THE INVESTIGATION ................................................................................. 19
    4.2.1 STEP 1: 1 MOTION CONTROL LOOP MAPPED ON 1 SPU ........................................... 21
    4.2.1.1 TIMING ANALYSIS .................................................................................................... 21
    4.2.1.2 RESOURCE USAGE ANALYSIS .................................................................................. 29
    4.2.2 STEP 2: MAX # OF MOTION CONTROL LOOPS THAT CAN BE MAPPED ON 1 SPU .... 31
    4.2.2.1 TIMING ANALYSIS .................................................................................................... 31
    4.2.2.2 RESOURCE USAGE ANALYSIS .................................................................................. 31
    4.2.3 STEP 3: MAX # OF SPUS THAT CAN BE CONNECTED TO 1 RINGBUS SYSTEM .......... 32
    4.2.3.1 TIMING ANALYSIS .................................................................................................... 32
    4.2.3.2 RESOURCE USAGE ANALYSIS .................................................................................. 33
    4.2.4 STEP 4: MAX # OF SPUS THAT CAN LOCATE ON 1 FPGA .......................................... 33
    4.2.4.1 TIMING ANALYSIS .................................................................................................... 33
    4.2.4.2 RESOURCE USAGE ANALYSIS .................................................................................. 33
  4.3 CONCLUSION OF THE INVESTIGATION ............................................................................... 36

5 ARCHITECTURE & DESIGN ......................................................................................................... 38
  5.1 SYSTEM LEVEL ARCHITECTURE ......................................................................................... 38
    5.1.1 FUNCTIONAL SPECIFICATION ....................................................................................... 38
    5.1.2 WORKING MECHANISM OF THE SYSTEM ..................................................................... 39
    5.1.3 BEHAVIORS OF THE SYSTEM COMPONENTS ................................................................. 41
    5.1.4 REALIZATION & DEPLOYMENT ....................................................................................... 42
  5.2 PROCESS CONTROLLER ARCHITECTURE ............................................................................ 43
    5.2.1 FUNCTIONAL SPECIFICATION ....................................................................................... 43
    5.2.2 DECOMPOSITION OF THE PROCESS CONTROLLER ..................................................... 44
    5.2.3 BEHAVIORS OF THE PROCESS CONTROLLER COMPONENTS .................................. 46
  5.3 MOTION SYSTEM ARCHITECTURE ....................................................................................... 47
    5.3.1 FUNCTIONAL SPECIFICATION ....................................................................................... 47
    5.3.2 DECOMPOSITION OF THE MOTION SYSTEM ............................................................... 48
    5.3.3 BEHAVIORS OF THE MOTION SYSTEM COMPONENTS .............................................. 50
  5.4 IO SYSTEM ARCHITECTURE ................................................................................................. 53
    5.4.1 FUNCTIONAL SPECIFICATION ....................................................................................... 53
    5.4.2 WORKING MECHANISM OF THE IO SYSTEM .............................................................. 53
    5.4.3 DECOMPOSITION OF THE IO SYSTEM ......................................................................... 54
5.12.3.1 BEHAVIORS OF DATA TRACER........................................................................................................103
5.12.3.2 BEHAVIORS OF SIGNAL INJECTOR ..............................................................................................104
5.12.3.3 BEHAVIORS OF ERROR LOGGER ..................................................................................................105
5.12.3.4 BEHAVIORS OF EVENT Handler ................................................................................................106
5.12.4 REALIZATION & DEPLOYMENT OF FACILITIES .........................................................................106
5.12.4.1 FUNCTIONAL COMPONENT DEPLOYMENT ...............................................................................107
5.12.4.2 INTERFACE DEPLOYMENT ........................................................................................................107

6 IMPLEMENTATION & REALIZATION .................................................................................................109
6.1 IMPLEMENTATION PROCEDURE ........................................................................................................109
6.2 POTENTIAL MODIFICATIONS ........................................................................................................110
6.3 WORKING MECHANISM ..................................................................................................................110

7 CONCLUSION .....................................................................................................................................114

APPENDIX A <LEGEND FOR SHAPES AND SIGNS> ..............................................................................116
APPENDIX B <LEGEND FOR SOME WORDS> .........................................................................................117
APPENDIX C <FUNCTIONAL OVERVIEW OF THE SYSTEM> ..............................................................119
APPENDIX D <ABBREVIATIONS> ........................................................................................................120

APPENDIX E: CARM ARCHITECTURE ..................................................................................................121
8.1 PROCESS CONTROL LAYER ...........................................................................................................122
8.2 TRANSDUCER LAYER .....................................................................................................................123
8.3 CONNECTIVITY LAYER ....................................................................................................................124

APPENDIX F: RINGBUS BASED MOTION CONTROLLER ......................................................................127

APPENDIX G: FPGA ..............................................................................................................................128

APPENDIX H <FSCR> ............................................................................................................................137

APPENDIX I <REFERENCE DOCUMENTS> ........................................................................................139
LIST OF FIGURES

Figure 1.1 The semiconductor process and the focus for ASML........................................... 8
Figure 2.1 Illumination with DOEs and FlexRay illumination ..................................................... 10
Figure 2.2 The Loop of MASU, FSCR and MMA ................................................................. 10
Figure 3.1 Calculation latencies for motion control loop ........................................................... 12
Figure 4.1 SPU Architecture ..................................................................................................... 15
Figure 4.2 ALU Instruction ........................................................................................................ 15
Figure 4.3 An Example Scenario for the motion control loop blocks .......................................... 17
Figure 4.4 RingBus Motion System ............................................................................................. 17
Figure 4.5 Example Scenario of the Input Handling in the RingBus Motion System .................... 18
Figure 4.6 Representation of 1 motion control loop is mapped on 1 SPU ................................. 19
Figure 4.7 Representation of max # motion control loops that can be mapped on 1 SPU ............ 20
Figure 4.8 Representation of max # of SPUs that are connected to the 1 Ringbus ....................... 20
Figure 4.9 Representation of max # of SPUs on 1 FPGA ............................................................ 21
Figure 4.10 Motion Controller Blocks in the current Implementation ........................................... 21
Figure 4.11 Example scenario for the motion control loop .......................................................... 22
Figure 4.12 The calculations in Block 1 ...................................................................................... 22
Figure 4.13 Operational view of Block 1 ................................................................................... 22
Figure 4.14 The calculations in Block 2 ..................................................................................... 23
Figure 4.15 Operational view of Block 2 ................................................................................... 23
Figure 4.16 Pre-calculations of Block 2 ..................................................................................... 24
Figure 4.17 Critical Path of Block 2 .......................................................................................... 24
Figure 4.18 Post-calculations of Block 2 .................................................................................... 24
Figure 4.19 Operational View of Block 3 .................................................................................. 25
Figure 4.20 Operational View of Block 4 .................................................................................. 25
Figure 4.21 Operational view of Block 5 ................................................................................... 25
Figure 4.22 The schedule of Block 2 and Block 4 together .......................................................... 26
Figure 4.23 The schedule of Block 2, Block 3 and Block 4 together .............................................. 27
Figure 4.24 The schedule of Block 1, Block 2, Block 3 and Block 4 together ................................. 27
Figure 4.25 The critical path of the final schedule ..................................................................... 28
Figure 4.26 The critical path of the motion control loop with the pre-calculations of Block 2 ....... 28
Figure 4.27 The final schedule of the motion control loop .......................................................... 28
Figure 4.28 Mapping 1 motion control loop on 1 SPU & SPU architecture ............................... 29
Figure 4.29 Representative overview of MMA .......................................................................... 30
Figure 4.30 Mapping more than one motion control loops on 1 SPU & SPU architecture .......... 31
Figure 4.31 Memory based SPU usage analysis ......................................................................... 34
Figure 4.32 Logic elements based SPU usage analysis ............................................................... 35
Figure 4.33 SPU usage analysis together .................................................................................... 36
Figure 5.1 System Level Architecture ....................................................................................... 38
Figure 5.2 State-Chart of the System ......................................................................................... 40
Figure 5.3 Sequence diagram of sending the parameters and the SPG profile to the Motion System 41
Figure 5.4 Sequence diagram of the real-time calculation in system level ................................... 41
Figure 5.5 Sequence diagram of the command handling in system level .................................... 42
Figure 5.6 Sequence diagram of sending error data, event data or traced data to the Process 42
Controller .................................................................................................................................. 42
Figure 5.7 Deployment of the System Level Architecture ............................................................ 43
Figure 5.8 The Process Controller Component ......................................................................... 44
Figure 5.9 The decomposition of the Process Controller ............................................................. 45
Figure 5.10 Sequence diagram of sending the parameters and the SPG profile to the Motion System 46
Figure 5.11 Sequence diagram of sending the target values to the Motion System ...................... 46
Figure 5.12 Sequence diagram of command handling in the Process Controller ......................... 46
Figure 5.13 Sequence diagram of data tracing and signal injection in the Process Controller ....... 47
Figure 5.14 The Motion System component .............................................................................. 48
Figure 5.15 The decomposition of the Motion System ................................................................... 48
Figure 5.16 Sequence diagram of sending the parameters and the SPG profile to the Mirror 48
Controller .................................................................................................................................. 50
Figure 5.17 Sequence diagram of real time calculation in the Motion System ............................ 51
An FPGA based motion controller

Figure 5.18  Sequence diagram of data tracing and signal injection in the Motion System .......................... 52
Figure 5.19  Sequence diagram of command handling in the Motion System ........................................ 52
Figure 5.20  IO System ..................................................................................................................... 53
Figure 5.21  State-Chart of the IO System ......................................................................................... 54
Figure 5.22  Decomposition of IO System ......................................................................................... 54
Figure 5.23  Sequence diagram of receiving sensor data and attaching ID by IO Module ....................... 56
Figure 5.24  Sequence diagram of sending out the actuator data by IO Module ................................. 56
Figure 5.25  Sequence diagram of receiving command by IO Module ............................................... 56
Figure 5.26  Sequence diagram of sending out the result by IO Module ............................................ 57
Figure 5.27  The interface deployment of IO System ............................................................................. 57
Figure 5.28  Dispatcher .................................................................................................................... 58
Figure 5.29  State-Chart of the Dispatcher ......................................................................................... 59
Figure 5.30  Decomposition of Dispatcher ......................................................................................... 59
Figure 5.31  Sequence diagram of dispatching the sensor data by Dispatcher Module ......................... 61
Figure 5.32  Sequence diagram of collecting the actuator data by Dispatcher Module ....................... 61
Figure 5.33  Sequence diagram of receiving the command by Queue ................................................... 62
Figure 5.34  Sequence diagram of receiving the result by Queue ....................................................... 62
Figure 5.35  Dispatcher Module ......................................................................................................... 63
Figure 5.36  Dispatcher Module decomposition .................................................................................... 63
Figure 5.37  Sequence diagram of dispatching the sensor data by Input Dispatcher ............................ 65
Figure 5.38  Sequence diagram of collecting the actuator data by Output Collector ............................ 65
Figure 5.39  Sequence diagram of dispatching the command by Input Dispatcher .............................. 65
Figure 5.40  Sequence diagram of collecting the result by Output Collector ....................................... 66
Figure 5.41  The interface deployment of Dispatcher Module ............................................................. 67
Figure 5.42  Queue ............................................................................................................................ 68
Figure 5.43  Queue decomposition ....................................................................................................... 68
Figure 5.44  Sequence diagrams of Queue components during initialization ........................................ 70
Figure 5.45  Sequence diagrams of Queue components for the commands during run time ................ 71
Figure 5.46  Interface deployment of the Queue .................................................................................. 72
Figure 5.47  The Mirror Controller Component .................................................................................. 73
Figure 5.48  The decomposition of the Mirror Controller ..................................................................... 73
Figure 5.49  Sequence diagram of sending the parameters and setpoint profile to the Internal
States&Parameters Handler .................................................................................................................. 76
Figure 5.50  Sequence diagram of the real time calculation in the Mirror Controller ............................ 76
Figure 5.51  Sequence diagram of data tracing and signal injection in the Mirror Controller ............... 77
Figure 5.52  Sequence diagram of command handling in the Mirror Controller ................................. 78
Figure 5.53  State-Chart of the Mirror Controller ................................................................................. 79
Figure 5.54  Deployment of the Mirror Controller ................................................................................ 80
Figure 5.55  Interface deployment of the Mirror Controller ................................................................. 81
Figure 5.56  Local IO System Component .......................................................................................... 82
Figure 5.57  The decomposition of the Local IO System ..................................................................... 83
Figure 5.58  Sequence diagram of receiving sensor data ..................................................................... 84
Figure 5.59  Sequence diagram of sending actuator data .................................................................... 84
Figure 5.60  Sequence diagram of injecting data ................................................................................. 85
Figure 5.61  Sequence diagram of tracing data .................................................................................... 85
Figure 5.62  Sequence diagram of command handling ......................................................................... 85
Figure 5.63  Interface deployment of the Local IO System .................................................................... 86
Figure 5.64  The Internal States & Parameters Handler Component ..................................................... 87
Figure 5.65  The decomposition of the Internal States & Parameters Handler ........................................ 88
Figure 5.66  Sequence diagram of parameters and setpoint storage in the Storage for Parameters ............................................................................................................................................. 89
Figure 5.67  Sequence diagram of providing the internal states and parameters to the Control Loop ............................................................................................................................................... 89
Figure 5.68  Sequence diagram of command handling in the Internal States&Parameters Handler ........ 90
Figure 5.69  Interface deployment of the Internal States Parameters Handler ....................................... 92
Figure 5.70  The Control Loop Component .......................................................................................... 92
Figure 5.71  The decomposition of the Control Loop ........................................................................... 93
Figure 5.72  Sequence diagram of real time calculation in the Control Loop ....................................... 95
Figure 5.73  Sequence Diagram of the real time calculation and data tracing in component level ........ 95
Figure 5.74  Sequence Diagram of error or event handling in component level .................................. 96
Figure 5.75  Functional Component deployment of the Control Loop .................................................. 96
Figure 5.76  Interface deployment of a block in the Control Loop ....................................................... 98
Figure 5.77  The communication details between two sequential components .................................... 98
Figure 5.78 Interface deployment of Block 3 in the Control Loop ........................................98
Figure 5.79 Interface deployment of the Sensor Interface in the Control Loop ...........................99
Figure 5.80 Interface deployment of the Actuator Interface in the Control Loop .........................99
Figure 5.81 Interface deployment of the Control Loop ..........................................................100
Figure 5.82 The Facilities Component ......................................................................................101
Figure 5.83 The decomposition of the Facilities ........................................................................102
Figure 5.84 Sequence Diagram of data tracing .......................................................................103
Figure 5.85 Sequence Diagram of data tracing .......................................................................104
Figure 5.86 Sequence diagram of signal injection .....................................................................105
Figure 5.87 Sequence diagram of signal injection .....................................................................105
Figure 5.88 Sequence diagram of error logging .......................................................................106
Figure 5.89 Sequence diagram of event handling ......................................................................106
Figure 5.90 Interface deployment of the Facilities ....................................................................108
Figure 6.1 Procedure for the implementation ...........................................................................109
Figure 6.2 The data word on the RingBus ..................................................................................111
Figure 8.1 Control Architecture Reference Model .................................................................121
Figure 8.2 Layered Structure of CARM ..................................................................................121
Figure 8.3 Layered Structure of CARM ..................................................................................122
Figure 8.4 Control Philosophy and its mapping to CARM model ............................................122
Figure 8.5 Communication between the controller and the process ........................................123
Figure 8.6 The conversion from SI to bits and/or vice versa .......................................................124
Figure 8.7 Positioning Connectivity in CARM ......................................................................124
Figure 8.8 Connectivity layer in its context ............................................................................125
Figure 9.1 The RingBus Based Motion Controller FPGA ..........................................................127
Figure 10.1 Stratix IV GX Chip view ......................................................................................128
Figure 10.2 Overview of a DSP Block ....................................................................................132
Figure 10.3 Basic Two-Multiplier Adder Building Block ..........................................................132
Figure 10.4 Stratix IV Full DSP Block Cascading Feature for FIR Structures .........................133
Figure 10.5 Half DSP Block Architecture ...............................................................................134
Figure 2.3 FSCR based on 5 Slots ATCA Rack ..........................................................................137
Figure 2.4 FSCR Base Interface .............................................................................................137
Figure 2.5 FSCR SRIO fabric network interface .....................................................................138

LIST OF TABLES

Table 4.1 Distribution of the parameters according to their types .............................................30
Table 4.2 Required number of FPGAs .....................................................................................34
Table 4.3 Required number of FPGAs .....................................................................................36
Table 4.4 Required number of FPGAs .....................................................................................37
Table 5.1 Communication details of the System Level Architecture .........................................39
Table 5.2 Communication details of the Process Controller ....................................................45
Table 5.3 Communication details of the Motion System ..........................................................49
Table 5.4 Communication details of IO System components ..................................................55
Table 5.5 Communication details of IO System components ..................................................60
Table 5.6 Communication details of Dispatcher Module .........................................................64
Table 5.7 Communication details of Queue ............................................................................69
Table 5.8 Communication details of the Mirror Controller .....................................................75
Table 5.9 Communication details in the Local IO System .........................................................84
Table 5.10 Communication details of the Internal States&Parameters Handler .......................88
Table 5.11 Communication details of the Control Loop ..........................................................94
Table 5.12 Communication details of the Facilities .................................................................103
Table 6.1 Data words of the example scenario ........................................................................112
Table 10.1 Stratix GX530K 1517 Device Features ..................................................................129
Table 10.2 Trimatrix Memory Capacity and Distribution in Stratix IV GX530K 1517 Device ....129
Table 10.3 Number of DSP Blocks in Stratix IV GX530K 1517 Device ...................................131
1 INTRODUCTION

ASML is the world’s leading provider of lithography systems for the semiconductor industry, manufacturing complex machines that are critical to the production of integrated circuits or Microchips. The Company is founded in 1984 and headquartered in Veldhoven, the Netherlands, with over 60 sales and service offices located worldwide.

The main functionality of these lithography systems is to support the semiconductor process. The semiconductor process, which is shown in Figure 1.1, consists of the following steps: slicing, polishing, material deposition or modification, photoresist coating, exposure (step and scan), developing and baking, etching and ion implementation, removing the photoresist, completed wafer, separation, and packaging. ASML focuses on the exposure (step and scan) step.

Figure 1.1 The semiconductor process and the focus for ASML

The products of ASML are composed of several components. One of the most critical parts of the products is the illumination part. In the illumination part, a new technology called FlexRay is introduced that enables clients of ASML to create laser beams in all desired shapes. It is composed of many tiny mirrors. For each mirror there is a controller to control the motion of the mirror. For reasons of accuracy, these controllers have to run at a high sample frequency. This assignment aims to design an architecture for an FPGA based motion controller for FlexRay.

The FPGA-based motion controller has software and hardware design constraints. With respect to the software design constraint, it should fit in the CARM reference model which is explained in Appendix E. With respect to the hardware design constraint, the current HPPCs that are the CPUs used for the controller are to be replaced in a dedicated rack called FSCR by one or more FPGA boards. The FSCR is the platform for the servo calculations of the motion controller which is illustrated in Chapter 2. Moreover, the third constraint concerns utilizing and reusing possible parts of a technology called the RingBus based Motion Controller. This motion controller is already implemented on an FPGA and ready to use within ASML. Because of the similarities with the calculation primitives of the control loops of RingBus based Motion Controller architecture and the motion control loops in our project, some components of the RingBus based Motion Controller are reused in our project.

In this document, first a preliminary investigation is done to see whether reusing some parts from the RingBus based Motion Controller is feasible or not. In this investigation, quantitative analyses are done on both the motion controller in our project and the RingBus based Motion Controller.
Controller by applying different mapping scenarios and analyzing the results. As a result of the preliminary investigation, we have concluded that reusing relevant components of the RingBus based Motion Controller satisfies given performance requirements. Then, an FPGA-based architecture is designed for the motion controller of FlexRay with respect to a stepwise refinement approach. A number of components of the RingBus based Motion Controller are used in our architecture based on the results of the preliminary investigation. Finally, the implementation procedure is explained and a roadmap is provided for future work.

This document is structured in the following manner: In Chapter 2 the FlexRay technology and the FSCR rack are introduced. The problem statement and requirements of own architecture are given in Chapter 3. Chapter 4 investigates the reuse of relevant parts of the RingBus based Motion Controller to the FPGA based FlexRay motion controller. Chapter 5 provides the architectural design and explains each step of the refinement approach in detail. Chapter 6 includes the implementation and realization of the design. In Chapter 7 conclusions are given. Moreover, Appendices A and B have legends to help the reader to easily read the report. Appendix C has the overview picture of the system and Appendix D contains the abbreviations. The CARM reference model is given in Appendix E. Appendices F and G explain the RingBus based Motion Controller and the FPGA which is to be used for the implementation respectively. Appendix I contains a list of references.

This report is written by Mehmet Fatih Ozturk and Alper Genc from Embedded Systems department of TU/e. We have common and separate contributions to the report. Common contributions include the literature study. In this study, the CARM reference model, the RingBus based Motion Controller, FlexRay and the FPGA which is to be used for the implementation are covered. In Chapter 4, based on the literature study, we conducted a preliminary investigation on the system. In this investigation, we have our own contributions. Alper Genc focused on the timing analysis of the system and Mehmet Fatih Ozturk analysed the memory usage of the system. In Chapter 5, the architecture and design are covered. Because we applied a stepwise refinement approach, we first decomposed the Motion System as shown in Appendix C and each of us focused on different components. Alper Genc worked on the Mirror Controller component and Mehmet Fatih Ozturk worked on the IO System and Dispatcher. In each step of the refinement, we always shared our knowledge to proceed on the right track.
2 FLEXRAY

The illuminator is a key part of any lithography optical system. It conditions the light from the source, and causes the light beam to take on a prescribed shape on the mask. This conditioning and shaping of the light is known as the pupil shape.

Different mask patterns work better with different pupil shapes making the pupil shape a key factor in ensuring a robust process window for low-k1 production – where process tolerances are approaching the limit of manufacturability. As chip features shrink, more complex pupil shapes are required. FlexRay makes it easier and quicker to create those shapes.

Current illuminator technology uses glass discs called diffractive optical elements (DOEs) to shape the light. For complex pupil shapes, these DOEs have to be custom designed and manufactured. By contrast, FlexRay uses a programmable array of thousands of individually adjustable micro-mirrors. It can create any pupil shape in a matter of minutes – eliminating the long cycle time associated with DOE design and fabrication and thus accelerating ramp to yield for low k1 designs. More information can be found in [1] and [2]. Figure 2.1 shows existing illumination with DOEs and FlexRay illumination together.

For FlexRay, feedback control of the mirror angles was chosen in order to guarantee fast and accurate mirror control. The MASU is the sensor of the feedback loop. It measures the angle of the many tiny mirrors and feeds them to a servo unit. It compares the measured angle to a setpoint angle, and based on the difference drives the mirrors. The servo algorithm runs in a dedicated rack, the FSCR which is shown in Figure 2.2. More information on FSCR can be found in Appendix H.
3 PROBLEM STATEMENT

As mentioned in the introduction part, there is an array of many tiny mirrors needed to be controlled in the illumination process. For each mirror there is a control loop running. Currently, calculations for these control loops are handled by CPUs. To handle so many control loops, many CPUs are required. Moreover, usage of high number of CPUs requires much rack space and power. Hence, increasing the speed for mirror controllers does not seem economically feasible with CPU based platforms.

The problem mentioned above is caused by the sequential structure of general purpose CPUs. Instead of only using a large number of CPUs, combination of general purpose CPU(s) and FPGA(s) is an option. Compared to CPUs, FPGAs have the following advantages:

- Possibility of large-scale parallelism providing high computational capacity.
- Low non-recurring engineering expenses compared with ASIC design.
- Flexible development.

The goal of the our project is to define a proper architecture for an FPGA-based motion controller for FlexRay. To achieve this, we will utilize a previous project which is the RingBus based Motion Controller. Because this project has been already implemented on FPGA, we will reuse some parts of it in our assignment. However, a preliminary investigation is necessary to see whether it is feasible to integrate some parts of the RingBus based Motion Controller to our architecture. Chapter 4 includes all steps of the preliminary investigation.

Moreover, the architecture should meet the requirements which are explained in the following. There are several requirements for an FPGA based motion controller. We can group these requirements as primary and secondary requirements which are shown below.

- The primary requirements concern:
  - design constraints,
  - performance requirements,
  - auxiliary functions.
- The secondary requirements concern:
  - maintainability,
  - extensibility,
  - cost of goods,
  - scalability,
  - reusability.

All possible solutions must meet the primary requirements. They are essential for the system to work. The secondary requirements are used as selection criteria, used to weigh different possible solutions. Therefore, we will only focus on the primary requirements at this phase.

3.1 DESIGN CONSTRAINTS

The FPGA-based solution for the motion controller should conform to FSCR & CARM hardware and software solutions. With respect to the hardware design constraint, the HPPC processor boards are to be replaced in FSCR by one or more FPGA boards. The FSCR rack is explained in Chapter 2. With respect to the software design constraint, the motion controller should satisfy CARM facilities. These facilities are explained in Appendix E.

3.2 PERFORMANCE REQUIREMENTS

There are a number of components in the motion control loop. The MASU sends the sensor data to the motion control loop and the MMA receives the actuator data from the motion control
loop. As mentioned before, there are many tiny mirrors need to be controlled. For each mirror, there is an individual motion control loop. The performance requirements are as follows:

- The calculation latency for each mirror should be less than “1 / f_mirror”. The calculation latency is the time between the inputs to the sensor interface and the outputs of the actuator interface. In Figure 3.1, it is represented by $t_2$.
- All motion control loops together should run at the frequency of $f_{desired}$. It means that the time between the input of the first mirror to the sensor interface and the output of the last mirror from the actuator interface, which is represented by $t_1$ in Figure 3.1, should be less than “1 / $f_{desired}$”.
- All calculations should be in a single precision floating point.

![Figure 3.1 Calculation latencies for motion control loop](image)

3.3 AUXILIARY FUNCTIONS

In the motion control loop, the auxiliary functions are data tracing, signal injection, error logging and event handling. Data tracing and signal injection are used to diagnose problems in the system. Error logging mechanism reports the errors and event handling mechanism generates an event to trigger certain actions.

Data tracing concerns observing the data on some specific points in the motion control loops. In the current system, there are a number of trace points in one motion control loop. It is required that a number of them should be supported at the same time at system level. For instance, there are $m$ trace points in total and at least $n$ of them should be supported at the same time at system level ($n < m$).

Signal injection concerns injecting a signal (i.e. white noise) to the motion control loop and observing the response of the motion control loop. Therefore, after signal injection, data tracing is used to observe the response of the system. The signal injection is to analyze the behavior of the motion control loop. The injection data are generated by a predefined set of signal generation components. They are noise, sine sweep, sine sweep log and constant generators.

The error handling mechanism of the motion control loop should report the errors to the error logger. In the motion control loops, there can be three types of errors. The first one is that the
sensor data may become invalid. The second error happens when the input of controller is outside the specified range. The third one concerns errors in actuators.

Besides the requirements which are explained above, this assignment requires a road map to go further with the implementation step after having a proper architecture for an FPGA-based motion controller for FlexRay. The implementation procedure and the roadmap are explained in Chapter 6. Furthermore, the architecture should fit in the current way of working at ASML and should conform to the standard software framework that has been developed within ASML.
4 PRELIMINARY INVESTIGATION

In this chapter, a preliminary investigation is done to see whether reusing some parts from the RingBus based Motion Controller is feasible or not. Investigation also combines the information with the FPGA we have and provide conclusions at the end.

In Chapter 2, FlexRay project is explained. In Appendix F, RingBus based Motion Controller is given in higher levels to show the similarities with FlexRay project. As already mentioned in Chapter 3, problem statement includes benefiting from the reusage of some parts from the RingBus based Motion Controller as a starting point. The reason behind is that there are a lot of similarities between these two projects. In addition, RingBus based Motion Controller is already implemented and tested successfully. Reusing some parts from the RingBus based Motion Controller both decreases the amount of time that will be spent for similar process in both projects. Furthermore, it increases the stability because the RingBus based Motion Controller is already tested and verified.

This investigation is done based on three aspects. One is the Control Loop of the FlexRay motion controller which is already explained in detail in Chapter 2. Control Loop is the most time critical part of the motion controller and it is a key aspect for the investigation. Second one is the FPGA that is ordered for the implementation. Third one is the components of the RingBus based Motion Controller that might be integrated into the FlexRay Motion Controller.

In Appendix G, the specifications of the FPGA that is going to be used for the implementation are given. ASML advise to work with Altera Stratix IV GX FPGA 530K in this project because it is currently available within the company. Hence, both the preliminary investigation and the implementation are done based on this FPGA. SPU and Ringbus Motion System are treated in detail in section 4.1. Approach for the investigation and steps are covered in section 4.2. Conclusion is included in section 4.3.

4.1 SPU & RINGBUS

SPU is the signal processing unit and RingBus is the communication bus of the RingBus based Motion Controller. The real time calculations are executed by the SPU. SPUs are located around the RingBus Motion System. The RingBus Motion System and the SPU will be explained in the following sub-sections. In our assignment, SPU is used as a processing unit and RingBus is used for the communication.

4.1.1 SPU

The SPU is essentially a small DSP module. It is used as a processing unit in the RingBus based Motion Controller. The SPU consists of the Instruction Memory, Controller, ALU, Parameter Memory, Temporary Registers, Input Buffer, Output Buffer and the interconnects between these elements as shown in Figure 4.1.
ALU is the arithmetic logic unit inside the SPU and the basic operations (multiplication, addition, subtraction, saturation) are executed in the ALU. In the configuration phase, the program is loaded in the instruction memory of the SPU. In runtime, the Controller manages the ALU to execute the instructions in the Instruction memory sequentially.

The program is a sequence of ALU instructions. The hexadecimal syntax of an ALU instruction is illustrated in Figure 4.2. In an ALU instruction, there are 16 digits each has 4 bits. The 3rd digit (from left) shows the number of clock cycles how long the instruction should wait to start after previous instruction. The 4th digit (from left) shows the corresponding operation (multiplication, addition, subtraction, etc.). The next four digits show the address location of the first operand in the memory. The next four digits show the address location of the second operand in the memory. The last four digits show the address location for the result.

For each operation, there should be a corresponding ALU instruction in the instruction memory. The SPU is programmed to perform the basic operations serially. Each basic operation can be
started on any clock cycle. That means, in each clock cycle a new ALU instruction in the instruction memory can be started.

The ALU needs several clock cycles to calculate the result of the operation. If the ALU pipeline delay is $p$, starting an operation on clock $n$ will produce a result on clock $n+p$. Similarly, an operation started on clock $n+1$ will produce a result on clock $n+p+1$. This means that all operations are pipelined. The ALU has a pipeline delay of 8 clocks for all basic operations in the RingBus based Motion Controller.

In the SPU, the Parameter Memory stores the parameters which are necessary for the calculations inside the ALU. The Temporary Registers store the results of the ALU instructions when they will be needed later on in the algorithm. Input Buffer stores the input data and the Output Buffer stores the output data.

As shown in Figure 4.1, the ALU has communication with the Controller, Temporary Registers, Parameter Memory, Input and Output Buffer. An ALU instruction is started by the Controller inside the ALU and ALU takes the input data from the Input Buffer, the corresponding parameters from the Parameter Memory, the corresponding internal states from the Temporary Registers and performs the calculation. Afterwards, the output data is sent to the Output Buffer or saved in Temporary registers by the ALU.

4.1.1.1 SCHEDULING MECHANISM

In the ALU, because of the ALU pipeline delay, an operation that needs the result of a previous operation can only be started $p$ or more clock cycles later. Therefore, there will be a chain of operations in the algorithm that relies on the result of the previous operation in the chain. This chain of operations is called “critical path”. The other operations (pre-calculations and post-calculations) can be executed in parallel with the critical path. Hence, the length of the schedule will be determined by the critical path. This scheduling mechanism will be explained in this section.

For the ALU operations, to clearly understand the critical path, pre-calculations and post-calculations, consider the blocks in Figure 4.3. There are three blocks “A, B and C” that are sequentially connected. Block B requires the output of Block A and Block C requires the output of Block B. Consider the calculations in Block B. The critical path of Block B is a chain of operations inside Block B such that the first operation of this chain needs the result of the last operation of the critical path of Block A. Similarly, the result of the last operation of this chain is required for the first operation of the critical path of Block C. Hence, the entire critical path for these three blocks can be calculated by only adding critical paths of the Blocks A, B and C sequentially.

In Block B, there may be some calculations which can be done without waiting the output of Block A. For instance they can be the operations which use the results from the previous iteration. These kinds of calculations are called as pre-calculations. The pre-calculations of Block B can be executed in parallel with the critical path of Block A.

In Block B, there may be some other calculations which are not used to calculate the output of Block B in that iteration. For instance they can be the operations such that the results of them will be used in the next iteration. These kinds of calculations are called as post-calculations. The post-calculations of Block B can be executed in parallel with the critical path of Block C. Therefore, the key idea to find the schedule of the entire motion control loop is first to figure out the critical path of each block, then merging them to handle the entire critical path and finally adding the pre-calculations and the post-calculations to the schedule.
4.1.2 RINGBUS

In the RingBus based Motion Controller, the RingBus is used for the communication. The RingBus is based on an Avalon Streaming Bus to carry the real time data. The blocks are located around the RingBus and each block has communication with the RingBus as shown in Figure 4.4.

The RingBus has 32 bits for the data and 8 bits for the channel information. Each block around the RingBus publishes or subscribes the data according to the channel bits. For publishing the data to the RingBus, each block has fixed channel mapping. That means, each block sends its data to a specific channel into the RingBus. For instance ADC controller publishes the data to the channel “00000001”, the first SPU publishes the data to the channel “00000010”, the second SPU publishes the data to the channel “00000011” and so on.

For taking the data from the RingBus, each block has a configurable subscribe matrix. According to the configuration of the subscribe matrix, each block can access each channel on the RingBus. The subscribe matrices are configured by the higher level software via an Avalon MM bus.
In Figure 4.4, the input data comes into the RingBus through the ADC controller. The ADC Controller publishes the data to its specific channel. Then the subscribe matrix of the corresponding SPU is configured for this specific channel by the higher level software. Then this SPU takes the input data and performs the corresponding calculations. Afterwards, the SPU publishes the output data to its specific channel. Then, the subscribe matrix of the DAC controller is configured to take the output data. Finally, DAC controller sends the output data.

Each block has a RingBus connector as shown in Figure 4.4. A data goes through the RingBus connector within one clock. Hence, each RingBus connector on the RingBus creates one clock latency. For instance, if there are 50 blocks around the RingBus (which means there are 50 RingBus connectors on the RingBus), a data word can complete one cycle on the RingBus with 50 clock cycles of delay.

When a block publishes the data to the RingBus, this data always completes one cycle on the RingBus. After the cycle on the RingBus, this data is deleted by the RingBus connector of the block which has published the data. Therefore, each block can see each data on the RingBus. This is one of the main advantages of the RingBus system.

Because of the situation which is explained in the above paragraph, data tracing and signal injection are very simple in the RingBus architecture. As shown in Figure 4.4, there is one Tracing block and one Injection block around the RingBus. Because each data is visible to each block around the RingBus, the Injection block only publishes the data to its specific channel of the RingBus and the Tracing block can take any data from the RingBus upon a request.

Here, we will investigate the latency of the RingBus. This is the latency due to the communication through the RingBus. This latency is caused by the RingBus connectors. As an example scenario, assume a RingBus motion system with 5 SPUs and an IO System around one RingBus which is shown in Figure 4.5. In each clock cycle, the IO System publishes the new data word to the RingBus. The first data word will go to the first SPU, the second data word will go to the second SPU and so on. The sixth data word will go to the first SPU again, the seventh data word will go to the second SPU and so on.
In Figure 4.5, the blue numbers represent the data words. The red line represents the RingBus and the boxes on the red line represent the RingBus connectors. In the above table, each row represents a different clock cycle and each column represents the corresponding path on the RingBus (the path just below this column). As shown in Figure 4.5, in the first clock, the first data word is published to the RingBus. In the second clock, the second data word is published to the RingBus and the first data word goes to the first SPU. By analyzing Figure 4.5, it is concluded that the first data word goes to the first SPU in the second clock, the second data word goes to the second SPU in the fourth clock and so on. Hence, N number of SPUs creates 2N clock cycles latency for receiving the input data words.

Assume one data word is executed within L clock cycles on one SPU. In that case, the first output word of the first SPU will be published to the RingBus in clock L+2, the first output of the second SPU will be published to the RingBus in clock L+4 and so on. The first output of the Nth SPU will be published to the RingBus in clock 2N+L.

### 4.2 APPROACH FOR THE INVESTIGATION

In Appendix G, detailed information about the FPGA, that is going to be used in the implementation, is given. Moreover, RingBus and SPU were explained in detail in section 4.1. In this section, the approach which is used for the investigation is explained. In each step of the approach, there are some analyses made. In our case, it is preferred to focus on timing and resource usage analysis for the investigation. The rationale behind is that timing and the resource usage are the bottlenecks of the FPGA based new illumination system mirror controller implementation.

In sub-section 4.2.1, analysis is going to be done for the case of 1 motion control loop is mapped on 1 SPU. Figure 4.6 shows the first step of the approach. Analysis will be done in two bases. First one is timing analysis. Second one is the resource usage analysis. There is a conclusion at the end of each sub-section which gives the results based on the combination of these two analyses.

![Figure 4.6](Image)

**Figure 4.6** Representation of 1 motion control loop is mapped on 1 SPU

In sub-section 4.2.2, investigation is looking for an answer of maximum number of motion control loops that can be handled by 1 SPU. Second step of the approach is represented in
Figure 4.7. Similarly, analysis will include timing and resource usage analysis. There is a conclusion at the end of the sub-section. It does not only conclude the result of analyses but also gives the comparison of the results of the first step and the second step.

In sub-section 4.2.2, maximum number of motion control loops that can be handled by 1 SPU is investigated. In sub-section 4.2.3, analyses are done for the case of maximum number of SPUs that are connected to the 1 Ringbus. This is the third step of the approach and the representation of this step can be found in Figure 4.8. Similarly, analyses will include timing and resource usage analysis.
In sub-section 4.2.4, investigation is done for finding maximum number of SPUs that can locate on 1 FPGA. Last step of the approach is represented in Figure 4.9. Similarly, analyses will include timing and resource usage analysis.

![Figure 4.9](image)

**Figure 4.9** Representation of max # of SPUs on 1 FPGA

### 4.2.1 STEP 1: 1 MOTION CONTROL LOOP MAPPED ON 1 SPU

In this sub-section, 1 motion control loop is mapped on 1 SPU case is taken into account for the investigation as the first step. It can be found how this step looks like in Figure 4.6. Analyses are done in two bases. First one is timing analysis. Second one is resource usage analysis. There is a conclusion at the end of the sub-section which gives the results based on the combination of these two analyses.

#### 4.2.1.1 TIMING ANALYSIS

The Motion Controller consists of 10 blocks in the current version of the implementation as shown in Figure 4.10. The aim of this project is to deploy the functionality of all these blocks inside the FPGA, even inside the SPU which is the processing unit of our implementation.

![Figure 4.10](image)

**Figure 4.10** Motion Controller Blocks in the current Implementation
To perform the timing analysis of the motion control loop blocks, the calculations of each block have been investigated. Afterwards, the calculations of each block have been scheduled on one SPU according to the scheduling mechanism which is explained in sub-section 4.1.1.1. Finally, these individual schedules have been combined and the final schedule of the entire motion control loop has been determined.

Here, in order to clearly explain the scheduling mechanism of the motion control loop, we will give an example scenario as shown in Figure 4.11. This scenario includes five blocks and a number of calculations in each block. Note that, these calculations are just given to illustrate the scheduling mechanism of the motion control loop calculations. The calculations and blocks in Figure 4.11 are totally different than the real motion control loop blocks.

![Example scenario for the motion control loop](image)

**Figure 4.11** Example scenario for the motion control loop

**Block 1:** There is a matrix multiplication in Block 1 which is shown in Figure 4.12.

\[
\begin{bmatrix}
\text{Out1}_1 \\
\text{Out1}_2
\end{bmatrix} = \begin{bmatrix}
a & b \\
c & d
\end{bmatrix} \ast \begin{bmatrix}
\text{In1}_1 \\
\text{In1}_2
\end{bmatrix}
\]

![The calculations in Block 1](image)

**Figure 4.12** The calculations in Block 1

For the matrix multiplication in that block, there should be four multiplications and two summations. The multiplications are independent and in each clock a new multiplication can be started. However, the summations should wait for the results of these multiplications. Hence, the schedule of Block 1 is shown in Figure 4.13.

![Operational view of Block 1](image)

**Figure 4.13** Operational view of Block 1
In Figure 4.13, the schedule of Block 1 is shown. There are four multiplications, two summations and the corresponding schedule in the figure. For each block, we will draw the schedule like in Figure 4.13. In this figure, the boxes in the schedule represent the clock cycles. The red numbers in these boxes represent the operations and each operation is assigned to its corresponding number with a blue arrow. When a red number is written in a box in the schedule, it means that the corresponding operation starts in that clock cycle. According to the ALU pipeline delay of the SPU, eight clock cycles later the result will be shown. The green numbers under the boxes of the schedule indicates that the result of the corresponding operation is ready at that clock cycle and later. Finally the red arrow on the schedule indicates that, all of the outputs of the corresponding block are ready after that time.

**Block 2:** There is a delayed sum operation for each input in Block 2 as shown in Figure 4.14.

The delayed sum operation consists of a critical path, some pre-calculations which can be done beforehand and some post-calculations which can be done after the critical path. To clearly observe the critical path of each delay element, we should redraw the operational view as in Figure 4.15.

In Figure 4.15, A and B are the values of In2_1 and In2_2 in the previous iteration respectively. Therefore, the calculations with these values can be done and the values C and D can be calculated before this block receives the inputs. They are called as pre-calculations. After C and D are calculated, the critical path is only two multiplications and two additions. Finally there are some post-calculations which are necessary for the next iteration. After the output values are calculated, A and B values should be updated. For the next iteration, A becomes equal to In2_1 and B becomes equal to In2_2. The pre-calculations, critical path and the post-calculations of Block 2 are shown in Figures 4.16, 4.17 and 4.18.
Block 3: The output of Block 3 is the weighted sum of the inputs: “Out3 = In3_1 * g + In3_2 * h”. For the weighted sum operation, there are two multiplications and one addition. The multiplications can be done in parallel. After the results of the multiplications are performed, the summation can be done. The operational view of Block 3 is shown in Figure 4.19.
Block 4: The output of Block 4 is the weighted sum of the inputs: "Out4 = In4_1 * i + In4_2 * j". For the weighted sum operation, there are two multiplications and one addition. The multiplications can be done in parallel. After the results of the multiplications are performed, the summation can be done. The operational view of Block 4 is shown in Figure 4.20.

Block 5: The outputs of Block 5 are the inputs multiplied with a gain: "Out5_1 = In5_1 * k; Out5_2 = In5_2 * l". There are two multiplications and they can be done in parallel as shown in Figure 4.21.
So far, we have figured out the critical path and the schedule of each block in Figure 4.11 by exploiting the pipelining. From now on, these individual schedules will be combined step by step by utilizing the parallelization between the blocks.

**Merging the Individual Schedules**

**Step 1 - Combine Block 2 and Block 4:** Figure 4.22 shows the schedules of Block 2 and Block 4 independently and the combined schedule of these two blocks. Because Block 4 should wait for the outputs of Block 2 to start its execution, the schedule of Block 4 is simply added to the tail of the schedule of Block 2. Note that, here we only consider the critical path of Block 2. The pre-calculations and the post-calculations of Block 2 will be added to the final schedule later.

**Step 2 - Combine Step 1 and Block 3:** Figure 4.23 shows the schedules of Step 1 and Block 3 independently and the combined schedule of these two. Because the calculations in these two are independent, they can be executed in parallel. In Figure 4.23, it is observable that the gaps of the schedule of Step 1 are filled with the calculations of the schedule of Block 3. By exploiting the parallelization between the blocks, we can create the schedule of this step as the same length as the schedule of Step 1.
Step 3 - Combine Step 2 and Block 1: Figure 4.24 shows the schedules of Step 2 and Block 1 independently and the combined schedule of them. Because Step 2 should wait for the outputs of Block 1 to start its execution, the schedule of Step 2 is simply added to the tail of the schedule of Block 1.

Step 4 - Combine Step 3 and Block 5: Figure 4.25 shows the final critical path of the entire motion control loop. Because Block 5 should wait for the outputs of Step 3, the schedule of Block 5 is simply added to the tail of the schedule of Step 3. Now, the critical path of the motion control loop is defined and from now on, the pre-calculations and the post-calculations of Block
2 will be added to the schedule. Note that the final schedule in Figure 4.25 consists of two lines and it can be read like reading a book: first line, then second line and so on…

![Diagram](image1)

**Step 5 - Add the Pre-calculations of Block 2:** In the schedule in Figure 4.26, we have added the pre-calculations of Block 2 to the critical path in Figure 4.25. If the schedules in Figure 4.25 and 4.26 are compared, it is seen that the length of these two schedules are exactly the same. Since, the pre-calculations are put to the gaps of the critical path.

![Diagram](image2)

**Step 6 - Add the Post-calculations of Block 2:** In the schedule in Figure 4.27, we have added the post-calculations of Block 2 to the schedule in Figure 4.26. These post calculations are put to the gaps at the end of the schedule of the motion control loop. The schedule in Figure 4.27 is the final schedule of the motion control loop which is shown in Figure 4.11.

![Diagram](image3)

**Results of Scheduling**

According to the schedule in Figure 4.27, one SPU can execute the motion control loop calculations within 59 clock cycles. Note that, it is not the schedule of the actual motion control
loop. This is an example scenario which is given in Figure 4.11. From now on, the investigations will be made according to this example scenario because of confidentiality.

Because the project team in ASML is continuously working on the motion control loops, in the future there may be some add-ons to the systems. To adapt these potential changes, we will leave some margin in our schedule. We will investigate the architecture by considering the period of the motion control loop as 100 clock cycles.

To sum up, one SPU can finish one motion control loop within 100 clock cycles for the given example scenario in Figure 4.11.

### 4.2.1.2 RESOURCE USAGE ANALYSIS

In this sub-section, resource usage analysis is done based on memory and logic elements usage. This analysis is conducted for the case implementation of 1 motion control loop on 1 SPU. The rationale behind is that memory and logic elements usage are the bottleneck for the resource usage. Hence, this investigation provides a good idea of the system for the corresponding case. Figure 4.28 shows the case implementation of 1 motion control loop on 1 SPU together with the SPU architecture in detail.

**Memory Usage:**

SPU is explained in detail in the sub-section 4.1.1. Memory usage of the SPU is sum of the memory usages of the components within the SPU. These are instruction memory, parameter memory, input buffer, output buffer, ALU and registers.

Among these components parameter memory is the most critical memory because there are many mirrors to be controlled and all of these parameters belong to the mirrors should be saved into the internal memory. The reason of only using internal memory is that SPU is designed to work with internal memory locations. Because SPU is preferred to be used without any modification in the first place, all memory blocks are considered as internal memory. For the future work, another investigation can be done by considering the usage of external memories.

As it is mentioned above memory usage analysis will be done based on the parameters that are used within the motion control loop. Figure 4.29 is not the original but a representative figure
which shows the general overview of the unit called MMA that has all tiny mirrors on it. As shown, MMA consists of several parts. In each part, a number of mirrors are located.

Partial structure of MMA is very important for the memory usage analysis. The reason behind is that there are three types of parameters in the motion control loop which are:

- parameters which is specific to the mirror
- parameters which are common for the mirrors within a specific part
- parameters which are common for all mirrors

These three types should be taken into the account for the memory usage calculation. In Table 4.1, one can find the distribution of the parameters according to their types within the motion control loop.

<table>
<thead>
<tr>
<th>Parameter type</th>
<th>Size for 1 motion control loop (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters which is specific to the mirror</td>
<td>X</td>
</tr>
<tr>
<td>Parameters which are common for the mirrors within a specific part</td>
<td>Y</td>
</tr>
<tr>
<td>Parameters which are common for all mirrors</td>
<td>Z</td>
</tr>
</tbody>
</table>

Table 4.1 Distribution of the parameters according to their types

In this sub-section, for step 1 of the approach, memory usage analysis is done based on the information given above. Moreover, for other steps of the approach idea will remain the same.

Based on the Table 4.1, memory usage in bytes for the case 1 motion control loop is mapped on 1 SPU is below.

**Memory usage of 1 motion control loop for step 1 = X + Y + Z**

If number of SPUs are denoted by N_SPU, total memory usage in bytes based on the case 1 motion control loop is mapped on 1 SPU is below.

**Total memory usage for step 1 = N_SPU * (X + Y + Z)**

where \( N_{SPU} = N_{mirror} \)
Logic Elements usage:
One SPU needs 177 LABs (Logic Array Blocks) and in these LABs there are 1440 ALMs (Adaptive Logic Modules). These 1,440 ALMs have a total of 2,880 possible ALUTs (Adaptive LookUp Table), but only 1,951 of them are used. Furthermore, 4 (18x18) DSP Blocks are required per SPU. Note that, one ALM is equal to 2 ALUTs and 2 Registers. 10 ALMs form 1 LAB for Stratix Family.

4.2.2 STEP 2: MAX # OF MOTION CONTROL LOOPS THAT CAN BE MAPPED ON 1 SPU

In this sub-section, investigation is looking for an answer of maximum number of motion control loops that can be handled by 1 SPU as a second step of the approach. This step is already represented in Figure 4.7. Similar to the first step, this one is also analyzed into two parts: Timing analysis and Resource usage analysis.

4.2.2.1 TIMING ANALYSIS

In sub-section 4.2.1.1, we have found that one SPU can execute the motion control loop calculations within 100 clock cycles. If the clock frequency of the SPU is \( f_{\text{spu}} \), the sample frequency of one motion control loop on one SPU is calculated as follows:

\[
\frac{f_{\text{control loop}}}{f_{\text{spu}}} = 100
\]

Hence, to satisfy the required frequency which is \( f_{\text{desired}} \), “\( f_{\text{control loop}} / f_{\text{desired}} \)” motion control loops can be implemented sequentially on one SPU. However, there will be some additional latency in the IO System and the RingBus architecture. Therefore we should also leave some margin while allocating the motion control loops to the SPUs. As a rough estimation, we can allocate 80% of the total time for motion control loop calculations and 20% of the total time for the communication in the IO System and the RingBus. According to this estimation, we will allocate approximately “\( f_{\text{control loop}} / f_{\text{desired}} \)"0.8” motion control loops on one SPU.

4.2.2.2 RESOURCE USAGE ANALYSIS

Memory Usage:
In this case there are more than 1 motion control loop is mapped on SPU and investigation is trying to find the maximum number of these motion control loops. Figure 4.30 shows the case implementation of maximum number of motion control loops that can be mapped on 1 SPU together with the SPU architecture in detail.

Figure 4.30 Mapping more than one motion control loops on 1 SPU & SPU architecture
If there are more than one motion control loops mapped on SPU, some of the parameter types can benefit from it. For instance, parameters which are common to the all mirrors can be stored once. There is no need to store more than one this type of parameter set in one SPU. Similarly, if the motion control loops mapped on same SPU are all from the same part of the MMA, there is also no need to reserve a memory place except one parameter set for the parameters which are common for the mirrors within a specific part.

\[ N_{\text{max on 1SPU}} \] represents the maximum number of motion control loops that can be mapped on 1 SPU. It is already defined in the memory usage analysis of the Step 1 that \( N_{\text{mirror}} \) stands for the total number of the mirrors and \( N_{\text{SPU}} \) represents the number of SPUs in the system.

The number of the motion control loops on 1 SPU can be found for this step.

\[ N_{\text{max on 1SPU}} = \frac{N_{\text{mirror}}}{N_{\text{SPU}}} \]

Based on the Table 4.1, memory usage in bytes for the case maximum number of motion control loop that can be mapped on 1 SPU is below. Here it is assumed that all the motion control loops mapped on the SPU are from the same part of the MMA.

**Memory usage of 1 SPU for step 2 = \( N_{\text{max on 1SPU}} \times X + Y + Z \)**

Total memory usage in bytes based on the case maximum number of motion control loop that can be mapped on 1 SPU is below.

**Total memory usage for step 2 = \( N_{\text{SPU}} \times (N_{\text{max on 1SPU}} \times X + Y + Z) \)**

**Logic Elements usage:**

The logic element usage of one SPU is independent of the number of motion control loops mapped on that SPU. Therefore, the logic element usage is the same as in sub-section 4.2.1.2.

### 4.2.3 STEP 3: MAX # OF SPUS THAT CAN BE CONNECTED TO 1 RINGBUS SYSTEM

In this sub-section, investigation is looking for an answer of maximum number of SPUs that can be connected to 1 Ringbus System as a third step of the approach. This step is represented in Figure 4.8. Similar to the first and second step, this step is also analyzed into two parts: Timing analysis and Resource usage analysis.

#### 4.2.3.1 TIMING ANALYSIS

In sub-section 4.2.2.1, we have found that one SPU can execute \((f_{\text{control loop}} / f_{\text{desired}})\times0.8\) motion control loops sequentially to satisfy the required frequency. Because we have \( N_{\text{mirror}} \) motion control loops, we should have \( N_{\text{mirror}} / ((f_{\text{control loop}} / f_{\text{desired}})\times0.8) \) SPUs such that each SPU will execute \((f_{\text{control loop}} / f_{\text{desired}})\times0.8\) motion control loops sequentially. For each motion control loop, the input data and the output data will be transferred to/from the corresponding SPU via the RingBus.

The idea will be like that: if we have M SPUs, the IO System will send the first M mirrors’ data to the SPUs within the first M clock cycles. The first SPU will execute the first mirror data, the second SPU will execute the second mirror data and so on.
According to the analysis in sub-section 4.1.2, the first mirror data goes to the first SPU in the second clock, the second mirror data goes to the second SPU in the forth clock and M\textsuperscript{th} mirror data goes to the M\textsuperscript{th} SPU in 2M\textsuperscript{th} clock. Each SPU will start its execution directly after receiving the corresponding data word from the RingBus. Because one motion control loop can be finished within 100 clock cycles on one SPU, in clock 102 the output data of the first SPU will be ready, in clock 104 the output data of the second SPU will be ready and in clock 100+2M the output data of M\textsuperscript{th} SPU will be ready.

Instead of publishing to the RingBus, each SPU will store the first output data word in its own output buffer. Since, between the clocks 101 and 100+M, the next M mirrors' input data word will be published to the RingBus. So, in clock 102, the (M+1)\textsuperscript{th} mirror data goes to the first SPU, in clock 104, the (M+2)\textsuperscript{th} mirror data goes to the second SPU and in clock 100+2M, the 2M\textsuperscript{th} mirror data goes to the M\textsuperscript{th} SPU. Therefore, the SPUs will never stop working.

After clock 100+2M, the outputs of the first M mirrors are sent to the IO System via the RingBus. By this way, there will not be a congestion of the input and output words on the RingBus. According to this idea, M SPUs will create only 2M clock cycles latency due to the communication on the RingBus.

4.2.3.2 RESOURCE USAGE ANALYSIS

In this step, investigation is done to see how many SPUs in maximum can be connected to the Ringbus. Therefore, resource usage analysis is not necessary for this case.

4.2.4 STEP 4: MAX # OF SPUS THAT CAN LOCATE ON 1 FPGA

In this sub-section, maximum number of RingBus systems that can locate on 1 FPGA is taken into account for the investigation as the last step. It can be found how this step looks like in Figure 4.9. Analyses are done in two bases. First one is timing analysis. Second one is resource usage analysis. There is a conclusion at the end of the sub-section which gives the results based on the combination of these two analyses.

4.2.4.1 TIMING ANALYSIS

In this step, investigation is done to see how many SPUs in maximum can be located on 1 FPGA. Therefore, timing analysis is not necessary for this case.

4.2.4.2 RESOURCE USAGE ANALYSIS

Memory Usage:
As already mentioned in memory usage analysis of step 3, if there are more than one motion control loops mapped on SPU, some of the parameter types can benefit from it. For instance, parameters which are common to the all mirrors can be stored once. There is no need to store more than one this type of parameter set in one SPU. Similarly, if the motion control loops mapped on same SPU are all from the same part of the MMA, there is also no need to reserve a memory place except one parameter set for the parameters which are common for the mirrors within a specific part.

Because of the reason explained above, memory usage is critical for the investigation of finding maximum number of SPUs that can locate on 1 FPGA. For this investigation, FPGA resources should also be taken into account. All the information is about the FPGA resources is referenced from Appendix G which includes the detailed information on the FPGA we have for the project.
Figure 4.31 shows the relation between number of motion control loops on 1 SPU and number of SPUs that can be mapped on 1 FPGA based on memory usage.

Note: In Figure 4.31, X is the point on X-axis which represents the number of control loops on 1 SPU. Y is the point on Y-axis that represents the number of SPUs mapped on 1 FPGA. X and Y together show the point where Y is number of SPUs that can be mapped on 1 FPGA where there are X control loops on 1 SPU.

It can be seen from the Figure 4.31 that if the number of motion control loops running on 1 SPU is small then number of FPGAs to satisfy the target is high. Three examples are pointed in the figure. For each point X represents the number of motion control loops mapped on 1 SPU. Y represents the number of motion control loops that can locate on 1 FPGA. For each X point, there is a ratio between Y points of green to blue curve. This ratio represents the minimum number of required FPGAs. Based on these points, the required number of FPGAs can be found for the example points in Table 4.2.

<table>
<thead>
<tr>
<th># of motion control loops on 1 SPU (X)</th>
<th>Required number of FPGAs (K)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X=20</td>
<td>K=3</td>
</tr>
<tr>
<td>X=40</td>
<td>K=2</td>
</tr>
<tr>
<td>X=60</td>
<td>K=2</td>
</tr>
</tbody>
</table>

Table 4.2 Required number of FPGAs

It can be concluded from memory usage analysis that required number of FPGAs is decreasing with the increasing number of motion control loops that is mapped on the same SPU.

**Logic Elements usage:**

Logic elements usage analysis below is done based on two things. One is the logic elements usage of 1 SPU which is already explained in detail in sub-section 4.1.1. Second one is the FPGA resources which are covered in Appendix G in detail.

- 1 SPU needs 177 LABs and in these LABs there are 1,440 ALMs.
These 1,440 ALMs have a total of 2,880 possible ALUTs, but only 1,951 of them are used.

4 DSP Blocks (18x18) per SPU are required.

1 ALM = 2 ALUTs + 2 Registers

1 LAB = 10 ALMs

Our FPGA (EP4SGX530K) has 212,480 ALMs which is equal to 424,960 ALUTs and dedicated logic registers.

212,480 ALMs form 21,248 LABs (10 ALMs = 1 LAB for Stratix Family).

512 DSP Blocks (18x18).

The Resource Usage of 1 SPU in our FPGA:

- 177 LABs → 177 / 21,248 = 0.83%
- 1,440 ALMs → 1,440 / 212,480 = 0.68%
- 1,951 ALUTs → 1,951 / 424,960 = 0.46%
- 4 DSP Blocks → 4 / 512 = 0.78%

For the resource usage, consider the worst case situation: 0.83%

So, our FPGA can include: 100 / 0.83 = 120 SPUs.

Figure 4.32 shows the relation between number of motion control loops on 1 SPU and number of SPUs that can be mapped on 1 FPGA based on logic elements usage.

Note: In Figure 4.32, X is the point on X-axis which represents the number of control loops on 1 SPU. Y is the point on Y-axis that represents the number of SPUs mapped on 1 FPGA. X and Y together show the point where Y is number of SPUs that can be mapped on 1 FPGA where there are X control loops on 1 SPU.

Ratio between green and blue curve represents the minimum number of required FPGAs. From the figure, based on logic elements usage analysis, it can be concluded that 1 FPGA is enough if the number of motion control loops mapped on the same SPU is between 35 and
number of all mirrors (N\textsubscript{mirror}). Required number of FPGAs depends on the ratio between curves if number of motion control loops mapped on SPU is smaller than 35. In Figure 4.32 area with yellow lines in it shows the case where more than 1 FPGA is required.

<table>
<thead>
<tr>
<th># of motion control loops on 1 SPU (X)</th>
<th>Required number of FPGAs (K)</th>
</tr>
</thead>
<tbody>
<tr>
<td>35 &lt; X &lt; N\textsubscript{mirror}</td>
<td>K=1</td>
</tr>
<tr>
<td>0 &lt; X&lt; 35</td>
<td>1 &lt; K &lt; inf</td>
</tr>
</tbody>
</table>

Table 4.3 Required number of FPGAs

4.3 CONCLUSION OF THE INVESTIGATION

In this chapter, investigation has been done based on two analyses. First one is timing analysis for several cases. Other one is resource usage analysis which is analyzed into two sub topics: memory usage analysis and logic elements usage analysis. To sum up the conclusions that are derived from each steps, the results from each of the analysis are included in Figure 4.33.

In Figure 4.33, red line represents the timing constraint which is satisfied on the left hand of the line. In other words, if more than 40 motion control loops are mapped on 1 SPU, timing requirements are no longer satisfied. Area with yellow lines in it represents the case where required number of FPGAs is calculated based on logic elements usage. Area with black lines in it shows the case where the number of FPGAs is calculated based on memory usage.

Several conclusions can be made from Figure 4.33. These are as follows:
- If more than 40 motion control loops are mapped on 1 SPU, timing requirements are no longer satisfied. It is a timing constraint.
- There are two other constraints that are important to determine required number of FPGAs that is enough to satisfy the requirements. They are memory usage constraint and logic elements usage constraint.
- It can be easily seen from the figure that memory usage is the bottleneck because ratio between green curve to blue curve is always larger than ratio between green curve to orange curve. Hence, memory usage is very critical for the investigation of required number of FPGAs.

Based on the conclusions above, required number of FPGAs is determined by memory usage constraint and timing constraint. Table 4.4 shows the results which consider both these constraints.

<table>
<thead>
<tr>
<th># of motion control loops on 1 SPU (X)</th>
<th>Required number of FPGAs (K)</th>
</tr>
</thead>
<tbody>
<tr>
<td>40 &lt; X &lt; N_mirror</td>
<td>Timing constraint is not satisfied!!</td>
</tr>
<tr>
<td>0 &lt; X &lt; 40</td>
<td>1 &lt; K &lt; inf</td>
</tr>
</tbody>
</table>

Table 4.4 Required number of FPGAs

It is logical to choose the number of motion control loops on 1 SPU as close as to the timing constraint. In conclusion, 2 FPGAs are enough to satisfy the both constraints.
5 ARCHITECTURE & DESIGN

In this chapter Architecture and Design is covered for FPGA based new illumination system motion controller. Step-wise refinement approach is adopted for the Architecture and Design stage. From system level, the components are refined step by step towards the FPGA platform. The function and interface of the component are described; then the component is decomposed to one level deeper sub-component; the behaviors of each sub-component are described. Then for each sub-component, above steps of refinement is carried out. This refinement stops until the further decomposition is not required. Then, the functional components and interfaces deployment are discussed. Appendix C shows the overview of the stepwise refinement approach for the system. It can be seen that the decomposition depth is 4.

In Appendix A and Appendix B, the legends are available which consist of the information that will help to easily read and understand the chapter. They include the meaning of shapes, colors, arrow types and some words respectively.

For each level of the architecture, there are two related sections. First one is functional specifications of the components. Other one is sequence diagrams that show the behavior of the components. For both sections, the same communication paths are represented by the same letters to provide consistency.

5.1 SYSTEM LEVEL ARCHITECTURE

This chapter discusses the architecture at system level. Firstly, the functional specification of the system is introduced. Then, the working mechanism of the system is explained. Afterwards, the decomposition of the system is described. After the decomposition, the behaviors of the components are mentioned. Finally, the realization and deployment are discussed.

5.1.1 FUNCTIONAL SPECIFICATION

The Motion System consists of a number of mirrors each having individual sensor data, actuator data and a motion control loop. The aim of the system is to move the mirrors to the desired target positions. The system should satisfy the properties of the CARM architecture which is explained in Appendix E.
The system level architecture is shown in Figure 5.1. There are Sensors, Actuators, a Motion System and a Process Controller in the system level architecture. Sensors sense the mirror positions and send the sensor data to the motion system. The Motion System performs the motion control loop calculations and generates the actuator data. Actuators actuate the mirrors according to the actuator data which is sent by the Motion System. The Process Controller sends the SPG profile, target values, parameters and the corresponding commands to the Motion System. The communication details of the System Level Architecture are shown in Table 5.1.

<table>
<thead>
<tr>
<th>Communication</th>
<th>From</th>
<th>To</th>
<th>Initiator</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Sensor</td>
<td>Motion System</td>
<td>Sensor</td>
<td>Sensor data</td>
</tr>
<tr>
<td>B</td>
<td>Motion System</td>
<td>Actuator</td>
<td>Motion System</td>
<td>Actuator data</td>
</tr>
<tr>
<td>C</td>
<td>Process Controller</td>
<td>Motion System</td>
<td>Process Controller</td>
<td>Parameters</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SPG profile</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Target values</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Command</td>
</tr>
<tr>
<td>D</td>
<td>Motion System</td>
<td>Process Controller</td>
<td>Motion System</td>
<td>Result</td>
</tr>
</tbody>
</table>

Table 5.1 Communication details of the System Level Architecture

The system should satisfy the properties of the CARM architecture. The CARM architecture consists of the Process Control Manager, Process Control feedback control system, Process Control software and Process Control software block components which are explained in Appendix E. The Process Controller contains the Process Control Manager and Process Control feedback control system components and the Motion System contains the Process Control software and Process Control software block components of the CARM architecture. The Process Control Manager component handles the initialization and termination of the Process Control facility and provides diagnostic facilities like data tracing and signal injection. The Process Control feedback control system component provides a queue for actions to be executed and manages the collection of parameters used by blocks.

The System Level Architecture which is shown in Figure 5.1 is the highest level of our architecture. The architecture will be refined and the decomposition of the Motion System and the decomposition of the components of the Motion System will be investigated. In each level of the architecture, first the functional specification of the component is explained. Then the component is decomposed into sub-components and the behaviors of each sub-component are described. Finally interface and functional component deployment is investigated.

5.1.2 WORKING MECHANISM OF THE SYSTEM

In this section the working mechanism of the system will be explained which is illustrated in Figure 5.2. In Figure 5.7, the deployment of the system is shown. The Motion System is deployed on the FPGA. When we switch the FPGA on, the system is in the terminated state. Firstly, the Process Controller sends the parameters and the SPG profile to the Motion System. The parameters and the SPG profile are stored in the internal memory of the FPGA. Afterwards, the Process Controller sends the control mode selection command and the Motion System goes into the closed loop mode. This phase is called as initialization and after the initialization the system is in the running state.
In the running state, firstly the Process Controller sends the target values to the Motion System. Afterwards, the Motion System receives the sensor data which includes the current positions of the mirrors. Then, the Motion System performs the motion control loop calculations and generates the actuator data according to the difference between the target values and the current positions of the mirrors. Note that, for the calculations the Motion System uses the parameters which are stored during initialization. The movement of a mirror is performed according to the SPG profile. Therefore, in each iteration the Motion System uses the corresponding setpoint on the SPG profile.

In the running state, there are three different periods which are important to capture the working mechanism of the system. They are the moving period, stable period and total period. In the above paragraph it is explained that rotating the mirrors to the desired positions is not instantaneous. Mirrors always reach their destinations in a number of steps which are defined by the setpoints on the SPG profile. That means, when the next target values come, the mirrors approach their destinations by using the setpoints on the profile. Finally, mirrors reach their destinations in a number of iterations. This number is equal to the number of setpoints on the SPG profile. After the mirrors reach their destinations, they remain there stable until the next target values come. From now on, we will call the time between the new target values come and the mirror reaches its destination as the moving period. The time between the mirror reaches its destination and the next target values come is called as the stable period. The summation of the moving period and the stable period is called as the total period.

After the calculations, the Motion System sends the actuator data to the actuators. After the actuators actuate the mirrors, the Motion System receives the sensor data of the mirrors again and the story in the above paragraph starts again. For each iteration, the Mirror Controllers use the next setpoint on the SPG profile. When all setpoints are used, the mirrors should be in their target positions theoretically. Afterwards, the system goes into the idle state. During the idle state, the Motion System keeps the mirrors in the stable position and waits for the next target
values. Whenever the next target values are received from the Process Controller, the story in the above paragraphs starts again with the first setpoint of the SPG profile.

During the running state, if the Process Controller sends a command to the Motion System, the Motion System executes this command and sends the result back to the Process Controller. There are a number of commands and some of them may occur only in idle state. On the other hand, some commands may occur during the calculation state. In that case, the Motion System stops the calculation, executes the command, sends the result and turns back to the calculation state again.

### 5.1.3 BEHAVIORS OF THE SYSTEM COMPONENTS

In this sub-section, letters that are used to represent communication paths in Figure 5.3-5.6 are consistent with the letters in Figure 5.1. This stands for all sub-sections which explain the behaviors of the components.

During initialization, the Process Controller sends the parameters and the SPG profile to the Motion System, which is shown in Figure 5.3.

![Sequence diagram of sending the parameters and the SPG profile to the Motion System](image)

**Figure 5.3** Sequence diagram of sending the parameters and the SPG profile to the Motion System

Figure 5.4 shows the sequence diagram of the real-time calculation on system level. First, the Process Controller sends the target values to the Motion System. Then, sensors send the sensor data to the Motion System. The Motion System performs the motion control loop calculations and sends the actuator data to the actuators.

![Sequence diagram of the real-time calculation in system level](image)

**Figure 5.4** Sequence diagram of the real-time calculation in system level

Figure 5.5 shows the sequence diagram of the command handling on system level. The Process Controller sends a command to the Motion System. The Motion System executes the command and sends the result back to the Process Controller.
Figure 5.5 Sequence diagram of the command handling in system level

Figure 5.6 shows the sequence diagram for error handling, event handling and data tracing on system level. The error data, event data and the traced data are sent to the Process Controller without a request command.

Figure 5.6 Sequence diagram of sending error data, event data or traced data to the Process Controller

5.1.4 REALIZATION & DEPLOYMENT

The current implementation of the motion system is managed by a CPU based environment. In the current system, MASU measures the mirror positions and MMA actuates the mirrors to the desired positions. These two parts remain same in our architecture. Sensors are mapped to MASU and actuators are mapped to MMA as shown in Figure 5.7.

The requirement of our project is to handle the motion control loops by one or more FPGAs. Therefore, the Motion System should be mapped to FPGAs. It is a design decision to keep the Process Controller on the Host Processor. Because there are complex sequential processes in the Process Controller, it is not efficient to deploy the Process Controller into the FPGA. Moreover, the FPGA resources will only be used for the motion control loop calculations to satisfy the required frequency.

Another design decision is to use SRIO (Serial RapidIO) in the Interconnect part. The alternative is using Ethernet. The rationale is that inside ASML, SRIO is commonly used for the real-time communication. Hence, the protocols of SRIO are already present on different levels to satisfy the communication. If we choose using Ethernet in the Interconnect part, we should also need the corresponding protocols on these levels. It will both complicate the implementation and increase the resource usage of the FPGA. Therefore, SRIO will be used in the Interconnect part.
5.2 PROCESS CONTROLLER ARCHITECTURE

In this chapter the Process Controller architecture is discussed. Firstly, functions of the Process Controller are introduced. Then, the decomposition of the Process Controller is explained. After this decomposition, the behaviors of the Process Controller components are described.

5.2.1 FUNCTIONAL SPECIFICATION

The Process Controller communicates with the Motion System as shown in Figure 5.8. The Process Controller contains the Process Control Manager and Process Control feedback control system components of the CARM architecture. Therefore, the function of the Process Controller is to support the functions of Process Control Manager and Process Control feedback control system which are explained in Appendix E. Briefly we can explain the functions of the Process Controller as follows:

- During initialization, the Process Controller sends the parameters and the SPG profile to the Motion System.
- It sends the target values to the Motion System.
- It sends the commands to the Motion System.
- It receives the results from the Motion System.
- It receives the error data, event data and trace data from the Motion System.
- It sends the injection data to the Motion System.
5.2.2 DECOMPOSITION OF THE PROCESS CONTROLLER

The decomposition of the Process Controller and the communication details are shown in Figure 5.9 and Table 5.2 respectively. The Process Controller consists of the Process Control Manager, Process Control feedback control system and Proxy component.

Rationale behind the decomposition:
According to the design decisions in sub-section 5.1.4, the Process Control Manager and Process Control feedback control system components of the CARM architecture are mapped to the Host Processor and the Process Control software and Process Control software block of the CARM architecture are mapped to the FPGA. The Process Control software component inside the FPGA should communicate with the Process Control Manager and Process Control feedback control system components inside the Host Processor. Therefore some modifications are required for the Process Controller. The current interfaces should be kept the same between Process Control software, Process Control Manager and Process Control feedback control system components. Therefore, a Proxy component is required in the Process Controller. The Proxy will translate the commands from the Process Controller to the Motion System. It receives and dispatches the results from the Motion System to the Process Controller. It is a design decision that the Proxy is deployed on the Host Processor. Since, it is easy to implement the system in this way. The higher level software will be invisible and the FPGA will communicate with the Proxy. Moreover, the FPGA resources will only be used for the real-time calculations.
An FPGA based motion controller

Figure 5.9 The decomposition of the Process Controller

<table>
<thead>
<tr>
<th>Communication</th>
<th>From</th>
<th>To</th>
<th>Initiator</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Process Control Manager or Process Control feedback control system</td>
<td>Proxy</td>
<td>Process Control Manager or Process Control feedback control system</td>
<td>Parameters SPG profile Target values Command Injection data</td>
</tr>
<tr>
<td>B</td>
<td>Proxy</td>
<td>Process Control Manager or Process Control feedback control system</td>
<td>Proxy</td>
<td>Result Traced data</td>
</tr>
<tr>
<td>C</td>
<td>Proxy</td>
<td>Motion System</td>
<td>Proxy</td>
<td>Parameters SPG profile Target values Command Injection data</td>
</tr>
<tr>
<td>D</td>
<td>Motion System</td>
<td>Proxy</td>
<td>Motion System</td>
<td>Result Traced data</td>
</tr>
</tbody>
</table>

Table 5.2 Communication details of the Process Controller

The functions of the Process Controller components are mentioned briefly below.

**Process Control Manager:**
- The Process Control Manager component handles the initialization and termination of the Process Control facility.
- It handles diagnostic facilities like data tracing and signal injection.

**Process Control feedback control system:**
- The Process Control feedback control system component provides a queue for the actions to be executed.
- It manages the collection of parameters used by blocks.

**Proxy:**
- The Proxy component translates the commands from the Process Controller to the Motion System.
- It receives and dispatches the results from the Motion System to the Process Controller.
5.2.3 BEHAVIORS OF THE PROCESS CONTROLLER COMPONENTS

During initialization, the Process Controller sends the parameters and the SPG profile to the Motion System which is shown in Figure 5.10.

At the beginning of each total period, the Process Controller sends the target values to the Motion System as shown in Figure 5.11.

Figure 5.12 shows the sequence diagram of sending a command and receiving the result from/to the Process Controller. The Process Control Manager component sends the command to the Proxy component and the Proxy component translates the command. Afterwards, it sends the command to the Motion System. The result is received in a similar way.

In the Motion System, there are a number of tracing points. After the Process Controller sends Start Tracing Command to the Motion System, the data on these points should be delivered to the Process Controller in each iteration. After the Process Controller sends Stop Tracing
Command to the Motion System, the Motion System finishes sending the trace data. Between the Start Tracing Command and Stop Tracing Command, the trace data is sent without any request from the Process Controller.

Moreover, there are a number of injection points and the injection data which is sent from the Process Controller should be delivered to the corresponding injection point in the Motion System. The communications for data tracing and signal injection are shown in Figure 5.13.

![Sequence diagram of data tracing and signal injection in the Process Controller](image)

**5.3 MOTION SYSTEM ARCHITECTURE**

In this chapter, the Motion System architecture is discussed. Firstly, the functions of the Motion System are introduced. Then, the decomposition of the Motion System is explained as well as the behaviors of the Motion System components.

**5.3.1 FUNCTIONAL SPECIFICATION**

The calculations for the motion control loops are performed in the Motion System. The Motion system takes the target values from the Process Controller, receives the sensor data from the Sensor, finishes the motion control loop calculations and sends the actuator data to the Actuator. The Motion System interacts with the Sensor, Actuator and Process Controller as shown in Figure 5.14. The functions of the Motion System are the followings:

- During initialization, the Motion System receives the parameters and the SPG profile from the Process Controller.
An FPGA based motion controller

- It receives the target values from the Process Controller.
- It receives the sensor data from the Sensor.
- It performs the motion control loop calculations.
- It sends the actuator data to the Actuator.
- It receives the commands from the Process Controller.
- It executes the commands.
- It sends the results to the Process Controller.
- It sends the error data, event data and trace data to the Process Controller.
- It receives the injection data from the Process Controller.

Figure 5.14 The Motion System component

5.3.2 DECOMPOSITION OF THE MOTION SYSTEM

The decomposition of the Motion System and the communication details are shown in Figure 5.15 and Table 5.3 respectively. The Motion System consists of an IO System, a Dispatcher and the Mirror Controllers. Their functions are mentioned briefly below.

Figure 5.15 The decomposition of the Motion System
IO System:
- The IO System provides communication to the outside world.
- It receives and sends out the data.
- It attaches mirror ID to the received sensor data.

Dispatcher:
- The Dispatcher dispatches the data to the Mirror Controllers.
- It collects the data coming from the Mirror Controllers.

Mirror Controller:
- During initialization, the Mirror Controller receives the parameters and the SPG profile from the Dispatcher.
- It receives the target values and sensor data from the Dispatcher.
- It performs the motion control loop calculations.
- It sends the actuator data to the Dispatcher.
- It receives the commands from the Dispatcher.
- It executes the commands.
- It sends the results to the Dispatcher.
- It sends the error data, event data and trace data to the Dispatcher.
- It receives the injection data from the Dispatcher.

<table>
<thead>
<tr>
<th>Communication</th>
<th>From</th>
<th>To</th>
<th>Initiator</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Sensor</td>
<td>IO System</td>
<td>Sensor</td>
<td>Sensor data</td>
</tr>
<tr>
<td>B</td>
<td>IO System</td>
<td>Actuator</td>
<td>IO System</td>
<td>Actuator data</td>
</tr>
<tr>
<td>C</td>
<td>Process Controller</td>
<td>IO System</td>
<td>Process Controller</td>
<td>Command Parameters Setpoints Target values Injection data</td>
</tr>
<tr>
<td>D</td>
<td>IO System</td>
<td>Process Controller</td>
<td>IO System</td>
<td>Result Traced data</td>
</tr>
<tr>
<td>E</td>
<td>IO System</td>
<td>Dispatcher</td>
<td>IO System</td>
<td>Sensor data Injection data</td>
</tr>
<tr>
<td>F</td>
<td>Dispatcher</td>
<td>IO System</td>
<td>Dispatcher</td>
<td>Actuator data Traced data</td>
</tr>
<tr>
<td>G</td>
<td>IO System</td>
<td>Dispatcher</td>
<td>IO System</td>
<td>Command Parameters Setpoints Target values</td>
</tr>
<tr>
<td>H</td>
<td>Dispatcher</td>
<td>IO System</td>
<td>Dispatcher</td>
<td>Result</td>
</tr>
<tr>
<td>I</td>
<td>Dispatcher</td>
<td>Mirror Controller</td>
<td>Dispatcher</td>
<td>Sensor data Injection data</td>
</tr>
<tr>
<td>J</td>
<td>Mirror Controller</td>
<td>Dispatcher</td>
<td>Mirror Controller</td>
<td>Actuator data Traced data</td>
</tr>
<tr>
<td>K</td>
<td>Dispatcher</td>
<td>Mirror Controller</td>
<td>Dispatcher</td>
<td>Command Parameters Setpoints Target values</td>
</tr>
<tr>
<td>L</td>
<td>Mirror Controller</td>
<td>Dispatcher</td>
<td>Mirror Controller</td>
<td>Result</td>
</tr>
</tbody>
</table>

Table 5.3 Communication details of the Motion System
### Rationale behind the decomposition:
Decomposition of Motion System is done basically to isolate the functionalities so that each of these functionalities can be implemented separately without influencing the rest of the system.

- **IO System** has the functionality to communicate with the outside world. It behaves like a gate. All data request, reception and submission are done by IO System.
- **Dispatcher** is responsible for routing of both real-time and non-real time data to the desired components within FPGA. Its functionality is to focus on data IDs and reference address which data belongs to.
- **Mirror Controllers** have a functionality to satisfy the computational part of the motion control system. All calculations related to the movement of the mirror are done in this component. Sensor data is the input for the Mirror Controller and actuator data is the output after a set of calculations.

Based on the functionalities of the Motion System which are explained above, it is decomposed into three components: IO System, Dispatcher and Mirror Controllers

### 5.3.3 BEHAVIORS OF THE MOTION SYSTEM COMPONENTS

During initialization, the parameters and the SPG profile are sent to the Mirror Controller which is shown in Figure 5.16.

![Sequence diagram of sending the parameters and the SPG profile to the Mirror Controller](image)

Figure 5.16  Sequence diagram of sending the parameters and the SPG profile to the Mirror Controller

Figure 5.17 shows the real-time calculation in the Motion System. At the beginning of each total period, the Process Controller sends the target values to the corresponding Mirror Controller through the IO System and Dispatcher. Afterwards, the IO System receives the sensor data and sends this sensor data to the corresponding Mirror Controller through the Dispatcher. In the Mirror Controller, if some error or event occurs, this error or event is sent to the Process Controller. Finally, the actuator data is sent to the actuator through the Dispatcher and IO System.
During real time calculation which is shown in Figure 5.17, data tracing or signal injection may occur. There are a number of tracing points in the Mirror Controller. After the Process Controller sends Start Tracing Command to the Mirror Controller, the data on these points should be delivered to the Process Controller in each iteration. After the Process Controller sends Stop Tracing Command to the Mirror Controller, the Mirror Controller finishes sending the trace data. Between the Start Tracing Command and Stop Tracing Command, the trace data is sent without any request from the Process Controller.

Moreover, there are a number of injection points and the injection data which is sent from the Process Controller should be delivered to the corresponding injection point in the Mirror Controller. The communications for data tracing and signal injection are shown in Figure 5.18.
Figure 5.18  Sequence diagram of data tracing and signal injection in the Motion System

Figure 5.19 shows the sequence diagram of the command handling on the Motion System level. Each command has an ID which indicates the corresponding motion control loop. According to the IDs of the commands, a command is sent to the corresponding Mirror Controller through the IO System and the Dispatcher.

Figure 5.19  Sequence diagram of command handling in the Motion System
5.4 IO SYSTEM ARCHITECTURE

This chapter discusses the IO System architecture. Firstly, functions of the IO System are introduced. Then, decomposition of the IO System is explained. After decomposition, behaviors of the IO System components are described. Finally, realization and deployment of the IO System is discussed.

5.4.1 FUNCTIONAL SPECIFICATION

IO System is responsible for providing the communication between Motion System and outside system. All data coming in and going out pass through IO System. The data that come into the Motion System are sensor data coming from Sensor block and commands coming from Process Controller. The data that go out are actuator data going to Actuator block and results of commands going to Process Controller. Communication between Process Controller and IO System is represented by “Command” and “Result”. In fact, “Command” has several types. Please check Appendix B to see all these types.

As shown in Figure 5.20, IO System interacts with Sensor, Actuator and Process Controller that are located outside the system. It also interacts with Dispatcher within the Motion System.

5.4.2 WORKING MECHANISM OF THE IO SYSTEM

In this section the working mechanism of the IO System will be explained which is illustrated in Figure 5.21. There are three states running in parallel as shown in the figure. The IO System handles the sensor data communication, actuator data communication and the communication with the Process Controller concurrently.

The IO System gets sensor data from Sensors. It attaches mirror IDs to the received sensor data and sends the sensor data to the Dispatcher. Concurrently, the IO System waits for the actuator data from the Dispatcher. Whenever it receives the actuator data, it sends this data to the actuators. At the same time, the IO System communicates with the Process Controller for
the commands and target values. The IO System receives the data from the Process Controller and sends this data to the Dispatcher.

![State-Chart of the IO System](image)

### 5.4.3 DECOMPOSITION OF THE IO SYSTEM

This part discusses the decomposition of IO System as shown in Figure 5.22. The communication details between blocks can be found in Table 5.4. IO System consists of:

- **IO module:**
  1. It provides communication with outside.
  2. It receives and sending out the data.
- **Input module:**
  1. It receives the sensor data.
  2. It attaches Mirror ID to the received sensor data.
- **Output module:**
  1. It gets the destination of the actuator data.
  2. It sends out the actuator data to Actuator.

![Decomposition of IO System](image)
Rationale behind the decomposition:
Decomposition of IO System is done basically to isolate the functionalities so that each of these functionalities can be implemented separately without influencing the rest of the system.

- IO Module has the functionality to communicate with the outside world. It behaves like a gate. All data reception and submission are done by IO Module.
- Input Module is responsible for incoming data. After the reception of the data, it also attaches an ID to it.
- Output Module is responsible for outgoing data. It transfers the data coming from the system which has a destination in the outside like Actuators and Process Controller.

Based on the functionalities of the IO System which are explained above, it is decomposed into three components: IO Module, Input Module and Output Module.

### 5.4.4 BEHAVIORS OF THE COMPONENTS OF THE IO SYSTEM

This part discusses the activities of the components of the IO System. There are 4 activities in the IO System. These activities are independent from each other and they should run in parallel.

Figure 5.23 shows the sequence diagram of receiving sensor data. Sensor sends the sensor data to IO Module in a certain frequency. IO Module passes the sensor data to the Input Module. Input Module has a functionality of attaching an ID to the sensor data. In Figure 5.15, it can be seen that there is more than one Mirror Controller set. Dispatcher needs and ID information to dispatch data correctly. That is why Input Module attaches ID to the sensor data before reaching Dispatcher.

<table>
<thead>
<tr>
<th>Communication</th>
<th>From</th>
<th>To</th>
<th>Initiator</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Sensor</td>
<td>IO Module</td>
<td>IO Module</td>
<td>Sensor data</td>
</tr>
<tr>
<td>B</td>
<td>IO Module</td>
<td>Actuator</td>
<td>IO Module</td>
<td>Actuator data</td>
</tr>
<tr>
<td>C</td>
<td>Process Controller</td>
<td>IO Module</td>
<td>Process Controller</td>
<td>Command Parameters Setpoints Target values</td>
</tr>
<tr>
<td>D</td>
<td>IO Module</td>
<td>Process Controller</td>
<td>IO Module</td>
<td>Result</td>
</tr>
<tr>
<td>E</td>
<td>IO Module</td>
<td>Input Module</td>
<td>IO Module</td>
<td>Sensor data</td>
</tr>
<tr>
<td>F</td>
<td>Input Module</td>
<td>Dispatcher</td>
<td>Input Module</td>
<td>Sensor data with ID</td>
</tr>
<tr>
<td>G</td>
<td>Output Module</td>
<td>IO Module</td>
<td>Output Module</td>
<td>Actuator data</td>
</tr>
<tr>
<td>H</td>
<td>Dispatcher</td>
<td>Output Module</td>
<td>Dispatcher</td>
<td>Actuator data</td>
</tr>
<tr>
<td>I</td>
<td>IO Module</td>
<td>Dispatcher</td>
<td>IO Module</td>
<td>Command Parameters Setpoints Target values</td>
</tr>
<tr>
<td>J</td>
<td>Dispatcher</td>
<td>IO Module</td>
<td>Dispatcher</td>
<td>Result</td>
</tr>
</tbody>
</table>

Table 5.4 Communication details of IO System components
Figure 5.23  Sequence diagram of receiving sensor data and attaching ID by IO Module

Figure 5.24 shows the sequence diagram of sending out the actuator data. Dispatcher first sends actuator data to the Output Module. Output Module gets destination from the actuator data and sends it to the IO Module. Finally, IO Module sends out the actuator data to the Actuator.

Figure 5.24  Sequence diagram of sending out the actuator data by IO Module

Figure 5.25 and Figure 5.26 shows the sequence diagram of receiving the command and sending the result respectively. Process Controller sends the command to the IO Module first. IO Module passes command to the Dispatcher directly.

The reason for command is passing directly to the Dispatcher is that command already has ID with it and there is no need to attach any ID on it. There is a difference between sensor data and command. Sensor is not capable of putting an ID to the sensor data. That is why it is done by IO System. On the other hand, Process Controller already has the information which command belongs to which mirror. It can attach ID to the command. It is also an advantage that ID attachment process is handled by Process Controller which locates outside the system. It means that IO System does not need to deal with attaching ID to the commands. It reduces the complexity and computational intensity on FPGA.

Figure 5.25  Sequence diagram of receiving command by IO Module
5.4.5 REALIZATION & DEPLOYMENT OF THE IO SYSTEM

This part discusses the interface deployment of IO System. Moreover, realization and performance analysis of IO System will be covered. Figure 5.27 shows the interface deployment of IO System. Between IO Module and Input module there are two pairs of Avalon MM Master and Slave interfaces. For the communication line E in Figure 5.22, Master is located on IO Module side, Slave on Input Module side. In addition, between IO Module and Output Module, there is only one pair of Master and Slave interfaces. For communication line G, Master is on Output Module part and Slave on IO Module part.

The interfaces between Input Module & Output Module and Dispatcher are Avalon ST interfaces. The rationale behind is that the communication is point to point communication. Moreover, Avalon ST interfaces have an ability to provide high performance. These two communication lines, F and H, are used for communication of real time data like sensor and actuator data.

The interfaces among IO Module and Dispatcher are two pairs of Avalon MM Master and Slave interfaces as shown in Figure 5.27.
5.5 DISPATCHER ARCHITECTURE

In this chapter, Dispatcher architecture is discussed. Firstly, functions of the Dispatcher are introduced. Then, decomposition of the Dispatcher is explained. After decomposition, behaviors of the Dispatcher components are described. Finally, realization and deployment of the Dispatcher is discussed.

5.5.1 FUNCTIONAL SPECIFICATION

Dispatcher is responsible for dispatching the data coming from IO System to the Mirror Controller sets. Similarly, it collects the data coming from Mirror controller sets and passes the data to the IO System.

The data that come into the Dispatcher are sensor data and command coming from IO System. Actuator data and results are also coming to the Dispatcher from Mirror Controller sets. The data that go out are actuator data and results of commands going to the IO System. Similarly, sensor data and command are going out to the Mirror Controller sets.

As shown in Figure 5.28, Dispatcher interacts with IO System and Mirror Controller which is both located within Motion system.

5.5.2 WORKING MECHANISM OF THE DISPATCHER

In this section the working mechanism of the Dispatcher will be explained which is illustrated in Figure 5.29. There are three states running in parallel as shown in the figure. The Dispatcher handles the sensor data communication, actuator data communication and the non-real time data communication concurrently. Note that non-real time data is the commands and target values.

The Dispatcher waits for sensor data from the IO System. Whenever it receives the sensor data, it determines the destination of this data with respect to the mirror ID. Then, it sends the data to the corresponding Mirror Controller. Concurrently, the Dispatcher waits for the actuator data from the Mirror Controllers. Whenever it receives the actuator data, it sends this data to the actuators. At the same time, the Dispatcher receives the commands and target values from...
the IO System. First, it determines the destination of this data with respect to the mirror ID. Then, it queues this data in the corresponding Queue. Afterwards, it sends the data to the Mirror Controller according to the mechanism in the Queue which is explained in Chapter 5.7.

![State-Chart of the Dispatcher](image)

**5.5.3 DECOMPOSITION OF THE DISPATCHER**

This part discusses the decomposition of Dispatcher as shown in Figure 5.30. The communication details between blocks can be found in Table 5.5. Dispatcher consists of:

- **Dispatcher module:**
  1. It dispatches the data to the Mirror Controllers.
  2. It collects the data coming from Mirror Controllers.

- **Queue:**
  1. It receives dispatched commands.
  2. It sends commands to the corresponding Mirror Controller set.
  3. It receives acknowledgment from Mirror Controller set.

![Decomposition of Dispatcher](image)
An FPGA based motion controller

<table>
<thead>
<tr>
<th>Communication</th>
<th>From</th>
<th>To</th>
<th>Initiator</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>IO System</td>
<td>Dispatcher Module</td>
<td>IO System</td>
<td>Sensor data</td>
</tr>
<tr>
<td>B</td>
<td>Dispatcher Module</td>
<td>IO System</td>
<td>Dispatcher Module</td>
<td>Actuator data</td>
</tr>
<tr>
<td>C</td>
<td>IO System</td>
<td>Dispatcher Module</td>
<td>IO System</td>
<td>Command Parameters</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Setpoints Target values</td>
</tr>
<tr>
<td>D</td>
<td>Dispatcher Module</td>
<td>IO System</td>
<td>Dispatcher Module</td>
<td>Result</td>
</tr>
<tr>
<td>E</td>
<td>Dispatcher Module</td>
<td>Mirror Controller</td>
<td>Dispatcher Module</td>
<td>Sensor data</td>
</tr>
<tr>
<td>F</td>
<td>Mirror Controller</td>
<td>Dispatcher Module</td>
<td>Mirror Controller</td>
<td>Actuator data</td>
</tr>
<tr>
<td>G</td>
<td>Dispatcher Module</td>
<td>Queue</td>
<td>Dispatcher Module</td>
<td>Command Parameters</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Setpoints Target values</td>
</tr>
<tr>
<td>H</td>
<td>Queue</td>
<td>Dispatcher Module</td>
<td>Queue</td>
<td>Result</td>
</tr>
<tr>
<td>I</td>
<td>Queue</td>
<td>Mirror Controller</td>
<td>Queue</td>
<td>Command</td>
</tr>
<tr>
<td>J</td>
<td>Mirror Controller</td>
<td>Queue</td>
<td>Mirror Controller</td>
<td>Result</td>
</tr>
<tr>
<td>K</td>
<td>Queue</td>
<td>Mirror Controller</td>
<td>Queue</td>
<td>Parameters</td>
</tr>
<tr>
<td>L</td>
<td>Mirror Controller</td>
<td>Queue</td>
<td>Mirror Controller</td>
<td>Result</td>
</tr>
<tr>
<td>M</td>
<td>Queue</td>
<td>Mirror Controller</td>
<td>Queue</td>
<td>Setpoints</td>
</tr>
<tr>
<td>N</td>
<td>Mirror Controller</td>
<td>Queue</td>
<td>Mirror Controller</td>
<td>Result</td>
</tr>
<tr>
<td>O</td>
<td>Queue</td>
<td>Mirror Controller</td>
<td>Queue</td>
<td>Target values</td>
</tr>
<tr>
<td>P</td>
<td>Mirror Controller</td>
<td>Queue</td>
<td>Mirror Controller</td>
<td>Result</td>
</tr>
</tbody>
</table>

Table 5.5 Communication details of IO System components

Rationale behind the decomposition:
Decomposition of Dispatcher is done basically to isolate the functionalities so that each of these functionalities can be implemented separately without influencing the rest of the system.
- Dispatcher Module has the functionality to dispatch the data which can be sensor data or command to the corresponding component.
- Queue is responsible for data coming from Process Controller. It does not deal with sensor or actuator data.

Based on the functionalities of the Dispatcher which are explained above, it is decomposed into two components: Dispatcher Module and Queue.
5.5.4 BEHAVIORS OF THE COMPONENTS OF THE DISPATCHER

This part explains the activities for the Dispatcher components. As it is seen in Figure 5.30, there are multiple communication paths available between Dispatcher Module and the Mirror Controller sets. Number of these separate communication paths is based on the number of separate Mirror Controller sets. In other words, for each Mirror Controller set, there is a Queue and two communication lines for sensor and actuator data. Sequence diagrams below represent the activities for just one set. Sequence diagrams for other Mirror Controller sets are exactly the same.

Activities can be analyzed mainly into two groups. First one is how Dispatcher components behave for sensor and actuator data. Second group is behaviors of the Dispatcher components for command and result data.

For sensor and actuator data, activities are not that complex. Figure 5.31 shows the sequence diagram of dispatching sensor data. IO System first sends the sensor data with ID attached to it to the Dispatcher Module. Dispatcher Module does some calculations based on the IDs to dispatch the data correctly. After this process, Dispatcher Module sends dispatched data directly to the corresponding Mirror Controller.

![Sequence diagram of dispatching the sensor data by Dispatcher Module](image1)

Figure 5.31 Sequence diagram of dispatching the sensor data by Dispatcher Module

Figure 5.32 shows the sequence diagram of collecting the actuator data from Mirror Controllers. Mirror Controllers send the actuator data when they finish their process. Dispatcher Module collects the data and passes it to the IO System.

![Sequence diagram of collecting the actuator data by Dispatcher Module](image2)

Figure 5.32 Sequence diagram of collecting the actuator data by Dispatcher Module

Figure 5.33 shows the behavior of the Queue for the data coming from Dispatcher Module. Queue receives the command and queues it and passes it to the corresponding component within Mirror Controller. In Figure 5.30, it can be realized that there are four outputs going out
from Queue. Sequence diagram shows the activity for one of them. In section 5.7, Queue is analyzed in detail for each output. It is similar with incoming data to the Queue from Mirror Controller. Figure 5.34 include the sequence diagram of receiving data from Mirror Controller and send it to the Dispatcher Module by Queue.

![Sequence diagram of receiving the command by Queue](image)

Figure 5.33 Sequence diagram of receiving the command by Queue

![Sequence diagram of receiving the result by Queue](image)

Figure 5.34 Sequence diagram of receiving the result by Queue

### 5.5.5 REALIZATION & DEPLOYMENT OF THE DISPATCHER

As it is already known by sub-section 5.5.2, Dispatcher is composed of Dispatcher Module and Queue. In sections 5.6 and 5.7, Dispatcher Module and Queue are analyzed respectively. Realization and deployment of the Dispatcher is covered by doing realization and deployment on both Dispatcher Module and Queue individually.

### 5.6 DISPATCHER MODULE

In this chapter, functional specification of the Dispatcher Module is firstly introduced. Decomposition of the Dispatcher Module and behaviors of the components follow the functional specification. Furthermore, realization and deployment of the module are presented at the end.

#### 5.6.1 FUNCTIONAL SPECIFICATION

The Dispatcher Module is one of the components within Dispatcher block. It is responsible for dispatching the data to and collecting the data from the corresponding components correctly. The data coming from IO System into the Dispatcher Module are sensor data and command. Moreover, actuator data is coming from set of Mirror Controllers and result of the command comes from Queue within Dispatcher. Dispatcher Module dispatches the data based on the ID that is attached to the data. In Realization part, dispatching technique, which is going to be used in the implementation, will be explained in detail.

As shown in Figure 5.35, Dispatcher Module interacts with IO System and set of Mirror Controllers outside the Dispatcher. It interacts with the Queue within the Dispatcher.
5.6.2 DECOMPOSITION OF THE DISPATCHER MODULE

In this part, decomposition of the Dispatcher Module is discussed. Figure 5.36 shows decomposition of the Dispatcher Module and Table 5.6 includes the communication details of the components of the Dispatcher Module. Dispatcher Module consists of:

- **Input Dispatcher:**
  1. It dispatches the sensor data to the corresponding set of Mirror Controllers.
  2. It dispatches the command data to the corresponding Queue.

- **Output Collector:**
  1. It collects the actuator data from all set of Mirror Controllers.
  2. It collects the result of the command from all Queues.

---

**Figure 5.35** Dispatcher Module

**Figure 5.36** Dispatcher Module decomposition
### Rationale behind the decomposition:
Decomposition of Dispatcher Module is done basically to isolate the functionalities so that each of these functionalities can be implemented separately without influencing the rest of the system.

- Input Dispatcher has the functionality to dispatch the incoming data which is sensor data to the corresponding Mirror Controller according to the ID that is already attached to the data. It only deals with incoming data.
- Output Collector has the functionality to collect the outgoing data which is actuator data from all Mirror Controllers. It only deals with outgoing data.

Based on the functionalities of the Dispatcher Module which are explained above, it is decomposed into two components: Input Dispatcher and Output Collector.

### 5.6.3 BEHAVIORS OF THE COMPONENTS OF THE DISPATCHER MODULE

In this part, activities of the behavior of the components of the dispatcher module are explained. There are mainly two activities in the Dispatcher Module. These activities are independent from each other and should run in parallel.

Behavior of the components can be analyzed into two parts. First part is receiving and dispatching the sensor data. Moreover, collecting the actuator data and pass it to the IO System is also analyzed in the first part. In the second part, dispatching the command coming from IO System and collecting the result from Queues is treated.

Figure 5.37 shows the dispatching of the sensor data. Input Dispatcher first receives the sensor data from the IO System. After getting the data, Input Dispatcher dispatches the sensor data based on the ID which is already attached the data. When dispatching process is done, based on the result, Input Dispatcher sends the sensor data to the corresponding Mirror Controller.
Figure 5.37 Sequence diagram of dispatching the sensor data by Input Dispatcher

Collection of the actuator data by Output Collector from Mirror Controllers is shown in Figure 5.38. Mirror Controller first sends the actuator data to the Output Collector. After Output Collector gets the data, it passes the data to the IO System.

Figure 5.38 Sequence diagram of collecting the actuator data by Output Collector

Figure 5.39 shows the dispatching of the command. Input Dispatcher first receives the command from the IO System. After getting the command, Input Dispatcher dispatches the command data based on the ID which is already attached. When dispatching process is done, based on the result of the process, Input Dispatcher sends the command to the corresponding Queue.

Figure 5.39 Sequence diagram of dispatching the command by Input Dispatcher

Collection of the result by Output Collector from Queues is shown in Figure 5.40. Queue first sends the result to the Output Collector. After Output Collector gets the result, it passes the result to the IO System.
5.6.4 REALIZATION & DEPLOYMENT OF THE DISPATCHER MODULE

This part discusses the interface deployment of Dispatcher Module. Moreover, realization and performance analysis of Dispatcher Module will be covered. Figure 5.41 shows the interface deployment of Dispatcher. Similar to the behaviors of the components section, deployment of the components is explained for the only one set. Other sets have exactly the same deployment; therefore it is enough to explain one set.

Between IO System and Input Dispatcher there is a one pair of Avalon MM Master and Slave interface and one pair of Avalon Streaming interfaces. There is a number of Avalon Streaming interface pairs between Input Dispatcher and Mirror Controller sets. The number that is mentioned above depends on the number of Mirror Controller sets. To give an example, if there are 5 sets of Mirror Controllers then there will be 5 pairs of Avalon Streaming interfaces between Input Dispatcher and Mirror Controller sets. In addition, there is one MM Master interface on the Dispatcher Module side and a number of MM Slave interfaces on the Mirror Controller sets side. The number of Slave interfaces depends on the number of Mirror Controller sets. According to [6], the best ratio from Master interface to Slave interface is 1:4. If number of Slaves increases, performance will decrease. That is why it is important to stick to the best ratio for the implementation.

Between IO System and Output Collector there is a one pair of Avalon MM Master and Slave interface and one pair of Avalon Streaming interfaces. There is a number of Avalon Streaming interface pairs and Avalon MM Master and Slave interface pairs between Output Collector and Mirror Controller sets. The number that is mentioned above depends on the number of Mirror Controller sets.
5.7 QUEUE

This chapter discusses the Queue architecture. Firstly, functions of the Queue are introduced. Then, decomposition of the Queue is explained. After decomposition, behaviors of the Queue components are described. Finally, realization and deployment of the Queue is discussed.

5.7.1 FUNCTIONAL SPECIFICATION

In the architecture which is shown in Figure 5.42, there are two types of communication between the IO System and the mirror controllers. The first one is the real time communication of the sensor and actuator data. The second type of communication between the IO System and the mirror controllers is for the commands, the target values and the non-real time data such as the setpoint profile and the parameters. It is done through the Queue which is responsible for the communication between the dispatcher module and the corresponding Mirror Controller. For each mirror controller, there is an individual Queue. As shown in Figure 5.42, the Queue interacts with five components: Dispatcher Module, Local IO System, Control Loop, Facilities and Internal States&Parameters Handler.
5.7.2 DECOMPOSITION OF THE QUEUE

The decomposition of the Queue and the communication details are shown in Figure 5.43 and Table 5.7 respectively. According to Figure 5.43, the Queue consists of the Queue Module and the Storage. The Queue module receives the input data from the dispatcher module and sends out this data to the storage. The Queue module also receives the result from the storage and sends out to the dispatcher module.

The storage communicates with the Queue module, the Local IO System, the Control Loop, the Internal States&Parameters Handler and the Facilities. It receives the data from the Queue module, temporarily stores and sends out this data. The storage also receives the result and sends out the result to the Queue Module.

<table>
<thead>
<tr>
<th>Communication</th>
<th>From</th>
<th>To</th>
<th>Initiator</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Queue Module</td>
<td>Dispatcher Module</td>
<td>Queue Module</td>
<td>Result</td>
</tr>
<tr>
<td>B</td>
<td>Dispatcher Module</td>
<td>Queue Module</td>
<td>Dispatcher Module</td>
<td>Command Target Values Parameter</td>
</tr>
</tbody>
</table>
Behaviors of the components of the Queue can be analyzed into two parts: Initialization and runtime.

Firstly the initialization command is sent to the Local IO System, the Control Loop, the Internal States&Parameters Handler and the Facilities. Afterwards the parameters and the setpoint profile are written to the Internal States&Parameters Handler through the Queue according to the “first in first out” mechanism. Then the control mode selection command is sent to the Control Loop and selects the nominal mode and the ON state. This entire part is called as initialization which is shown in Figure 5.44.
Figure 5.44 Sequence diagrams of Queue components during initialization

Figure 5.45 shows the run time activities of the Queue components for the data coming from Dispatcher Module. Queue Module receives the control mode command from the dispatcher module and sends this command to the Storage first. Storage then forwards the command to the Control Loop and the Internal States&Parameters Handler. During the initialization, control mode command selects the nominal mode and goes to ON state. Furthermore, Queue receives the parameters from the dispatcher module. Storage passes these parameters to the Internal States&Parameters Handler after getting them from Queue Module which is illustrated in Figure 5.45.

Another function of the Queue is receiving the new target values from the dispatcher module and sending it to the Control Loop. Additionally, Queue receives the Setpoint Profile from the dispatcher module and sends it to the Internal States&Parameters Handler.
5.7.4 REALIZATION & DEPLOYMENT OF THE QUEUE

In our architecture, because the communication of the commands and the target values are both via the Queue, an arbitration mechanism is required for the storage of the Queue. The parameters and the setpoint profile are written to the Internal States&Parameters Handler once while initialization. Therefore, “first in first out” mechanism can be used in the Queue for the parameters and the setpoint profile.

When we switch on the FPGA, firstly the initialization command is sent to the Local IO System, the Control Loop, the Internal States&Parameters Handler and the Facilities. Afterwards the parameters and the setpoint profile are written to the Internal States&Parameters Handler through the Queue according to the “first in first out” mechanism. Then the control mode selection command is sent to the Control Loop and selects the nominal mode and the ON state.

After the initialization, SPUs are ready to start. Because they are data triggered, they are waiting for the first target values. Whenever the first target values come to the SPUs, they start to execute. These target values are also communicated via the Queue and according to the “first in first out” mechanism.

In the current architecture, after the motion control loop starts its execution, the control loop mode does not change. Therefore, after the initialization, data which communicated through the Queue are only the target values; hence the “first in first out” arbitration mechanism supports
this communication. However, we will consider the scenario that the control mode selection command may occur during the runtime.

If the control mode selection command is received in runtime, all target values in the storage of the Queue which are received before the control mode selection command should be switched to zero. It means that, before the control mode selection command is sent to the Control Loop, all mirrors should be positioned to the zero position for safety. After all mirrors are in zero position, the control mode selection command is sent to the Control Loop. Then the Control Loop will continue with the next target values in the new mode.

**INTERFACE DEPLOYMENT**

Figure 5.46 shows the deployment of the Queue. The interface with the Dispatcher module is Avalon MM Slave interface. The Queue is passive and waits for the data from the dispatcher module. The interface with the Local IO System, the Control Loop, The Internal States&Parameters Handler and the Facilities is Avalon MM Master interface. That means Queue takes initiatives in these communications.

![Interface deployment of the Queue](image)

Figure 5.46 Interface deployment of the Queue

### 5.8 MIRROR CONTROLLER ARCHITECTURE

In this chapter, the Mirror Controller architecture is discussed. Firstly, the functions of the Mirror Controller are introduced. Then, the decomposition of the Mirror Controller is explained. After the decomposition, the behaviors of the Mirror Controller components are described. Afterwards, the working mechanism of the system on Mirror Controller level is expressed. Finally, the realization and deployment are discussed.

#### 5.8.1 FUNCTIONAL SPECIFICATION

The calculations of the motion control loops are performed in the Mirror Controller. The Mirror Controller interacts with the Dispatcher as shown in Figure 5.47. The functions of the Mirror Controller are the followings:

- During initialization, the Mirror Controller receives the parameters and the SPG profile from the Dispatcher.
- It receives the target values from the Dispatcher.
- It receives the sensor data from the Dispatcher.
- It performs the motion control loop calculations.
- It sends the actuator data to the Dispatcher.
- It receives the commands from the Dispatcher.
- It executes the commands.
- It sends the results to the Dispatcher.
- It sends the error data, event data and trace data to the Dispatcher.
- It receives the injection data from the Dispatcher.
As mentioned above, there are two basic functionalities of the Mirror Controller. They are the real time calculation and command handling. The Mirror Controller starts its execution with an initialization command. Afterwards, the Mirror Controller performs the real time calculation and command handling concurrently.

5.8.2 DECOMPOSITION OF THE MIRROR CONTROLLER

The decomposition of the Mirror Controller is shown in Figure 5.48. The Mirror Controller consists of the Local IO System, Control Loop, Internal States&Parameters Handler and Facilities.

Rationale behind the decomposition:
The Local IO System provides communication to the outside of the Mirror Controller. It is isolated to decouple the components from how the data are communicated with the outside system. The Control Loop is mainly responsible for the motion control loop calculations. The Internal States&Parameters Handler provides the internal states and parameters to the Control Loop. The Facilities is a group of functions required in the CARM architecture. Basically, the rationale behind the decomposition of the Mirror Controller is to isolate the functionalities of the components. Hence, a component can be changed or refined without affecting the rest of the system.
The functions of the Mirror Controller components are mentioned briefly below.

**Local IO System:**
- The Local IO System receives the sensor data from the Dispatcher.
- It sends the sensor data to the Control Loop.
- It receives the actuator data from the Control Loop.
- It sends the actuator data to the Dispatcher.
- It receives the injection data from the Dispatcher.
- It sends the injection data to the Facilities.
- It receives the tracing data from the Facilities.
- It sends the tracing data to the Dispatcher.
- It communicates with the Dispatcher for the corresponding commands.

**Control Loop:**
- The Control Loop receives the sensor data from the Local IO System.
- It performs the motion control loop calculations.
- It sends the actuator data to the Local IO System.
- It sends the mirror ID to the Internal States&Parameters Handler.
- It receives the parameters, internal states, setpoints and the target values from the Internal States&Parameters Handler.
- It sends back the internal states to the Internal States&Parameters Handler.
- It receives the injection data from the Facilities.
- It sends the tracing data to the Facilities.
- If some error or event occurs, it sends this error or event to the Facilities.
- It communicates with the Dispatcher for the corresponding commands.

**Facilities:**
- The Facilities receives the injection or tracing command from the Dispatcher.
- It receives the injection data from the Local IO System.
- It sends the injection data to the Control Loop.
- It receives the tracing data from the Control Loop.
- It sends the tracing data to the Local IO System.
- It logs the errors generated in the Control Loop.
- It handles the events generated in the Control Loop.
- It communicates with the Dispatcher for the corresponding commands.

**Internal States&Parameters Handler:**
- The Internal States&Parameters Handler receives the parameters, SPG profile and the target values from the Dispatcher.
- It provides the parameters, setpoints and the target values to the Control Loop.
- It receives the internal states from the Control Loop.
- It provides the internal states to the Control Loop.
- It communicates with the Dispatcher for the corresponding commands.

There are two different communications in the Mirror Controller. For the real time calculations, the Dispatcher sends the sensor data and receives the actuator data to/from the Mirror Controller through the Local IO System. However, the commands are directly sent to the corresponding component by the Dispatcher. Hence, each component has a communication with the Dispatcher in Figure 5.48. The details of the communications inside the Mirror Controller are shown in Table 5.8.
### 5.8.3 BEHAVIORS OF THE MIRROR CONTROLLER COMPONENTS

The basic idea of the Mirror Controller architecture is that the parameters, SPG profile and the target values should be available inside the Internal States&Parameters Handler while the Control Loop performs its calculations. Therefore, during initialization all parameters and the SPG profile should be stored in the Internal States&Parameters Handler as shown in Figure 5.49. The Dispatcher sends the parameters and the SPG profile to the Internal States&Parameters Handler. Moreover, at the beginning of each total period, the target values are sent to the Internal States&Parameters Handler in a similar way.
As mentioned in the functional specification part, there are two basic functionalities of the Mirror Controller. First one is the real time calculation which is shown in Figure 5.50. In the real time calculation, the Local IO System receives the sensor data from the Dispatcher and sends this data to the Control Loop. The Control Loop gets the corresponding parameters, internal states, setpoints and the target values from the Internal States&Parameters Handler according to the mirror ID of this sensor data. Then the Control Loop performs the motion control loop calculations. After the calculations it sends the internal states back to the Internal States&Parameters Handler. During the calculations, if an error or event occurs, the Control Loop sends this error or event to the Facilities. Finally the Control Loop sends the actuator data to the Local IO System. The Local IO System sends the actuator data to the Dispatcher. Moreover, the Control Loop sends the mirror ID to the Facilities to inform the Signal Injector inside the facilities about finishing the calculations of this mirror. This is to eliminate the risk that the data is injected halfway of a mirror calculation.

During real time calculation which is shown in Figure 5.50, data tracing or signal injection may occur. There are a number of tracing points in the Control Loop. After the Dispatcher sends Start Tracing Command to the Control Loop, the data on these points should be delivered to the Facilities in each iteration. The Facilities will send the trace data to the Dispatcher through the Local IO System. After the Dispatcher sends Stop Tracing Command to the Control Loop, the Control Loop finishes sending the trace data. Between the Start Tracing Command and Stop Tracing Command, the trace data is sent without any request.

Moreover, there are a number of injection points in the Control Loop and the injection data which is sent from the Dispatcher should be delivered to the corresponding injection point in the
An FPGA based motion controller

Control Loop. The injection data is sent to the Facilities through the Local IO System. Then the Facilities sends the trace data to the Control Loop. The communications for data tracing and signal injection are shown in Figure 5.51.

![Sequence diagram of data tracing and signal injection in the Mirror Controller](image)

Figure 5.51 Sequence diagram of data tracing and signal injection in the Mirror Controller

The second functionality of the Mirror Controller is the command handling. The delivery of the commands to the Mirror Controller components is not through the Local IO System. This is a design decision that the commands will be delivered to the Mirror Controller components from the Dispatcher via an Avalon MM bus. The rationale is to isolate the real time communication from the command handling, because we do not want to occupy the RingBus with the non-real time data. Although the real time communication is via the Local IO System, a component (QUEUE) inside the Dispatcher is responsible for the non-real time communication. The Dispatcher sends commands to the Internal States&Parameters Handler, Control Loop, Facilities and the Local IO System directly.

Figure 5.52 shows the command handling in the Mirror Controller. A command which comes from the Dispatcher may belong to the Local IO System, Control Loop, Internal States&Parameters Handler or Facilities. Hence, the commands in the Mirror Controller can be grouped under these four categories. Moreover, some commands require a result or acknowledge as shown in the figure.
5.8.4 WORKING MECHANISM OF THE SYSTEM ON MIRROR CONTROLLER LEVEL

In sub-section 5.1.2, the working mechanism of the system is explained. In this part, the working mechanism is explained on the Mirror Controller level. When we switch the FPGA on, the system is in the terminated state. During initialization, the Process Controller sends the parameters and the SPG profile to all Mirror Controllers through the IO System and Dispatcher. The parameters and the SPG profile are stored in the Internal States&Parameters Handler. After the initialization the system is in running state.

In the running state, firstly the Process Controller sends the target values to the Mirror Controllers. The Dispatcher dispatches each target value to the corresponding Mirror Controller. The target values are stored in the Internal States&Parameters Handler. Note that, the target values are sent at the beginning of each total period only once. In the remaining iterations of this total period, the stored target values are used. In the next total period, the new target values are overwritten to the Internal States&Parameters Handler.

After the target values are sent, the IO System receives sensor data of the mirrors and gives this data to the Dispatcher. The Dispatcher dispatches the sensor data to the corresponding Mirror Controllers. The Local IO System receives the sensor data from the Dispatcher and sends to the Control Loop. Whenever the Control Loop receives the sensor data, it starts the calculations. For the calculations, the Control Loop uses the target values and the setpoints which are stored in the Internal States&Parameters Handler.

After the calculations, the Control Loop sends the actuator data to the actuators through the Local IO System, Dispatcher and IO System. After the actuators actuate the mirrors, the Motion System receives the sensor data of the mirrors again and the story in the above paragraph starts again.
In each iteration, the Mirror Controllers use the next setpoint on the SPG profile. When all the setpoints are used, the mirrors should be in their target positions theoretically. Afterwards, the system goes into the idle state. During the idle state, the Motion System keeps the mirrors in the stable position and waits for the next target values. Whenever the next target values are received from the Process Controller, the story in the above paragraphs starts again.

During the running state, if the Process Controller sends a command to a Mirror Controller, this command is sent to the corresponding component of the Mirror Controller through the IO System and Dispatcher. The component executes this command and sends the result back to the Dispatcher. The Dispatcher sends the result to the IO System and the IO System sends it to the Process Controller.

During the running state, data tracing or signal injection may also occur. If the Process Controller sends a tracing command to the Facilities, the Facilities forwards this command to the Control Loop and the Control Loop starts to publish the trace data. The Facilities receives the trace data and stores it. The Facilities sends the trace data to the Process Controller through the Local IO System, Dispatcher and IO System when it is requested. If the Process Controller sends an injection data to the Facilities, the Facilities sends this data to the Control Loop.

In Figure 5.53, the state-chart of the Mirror Controller is shown. During initialization parameters and SPG profile are stored. After initialization new target value storage, real-time calculation and command handling are executed simultaneously.
5.8.5 REALIZATION & DEPLOYMENT OF THE MIRROR CONTROLLER

In this part, firstly the functional component deployment is introduced. Then, the interface deployment is explained.

5.8.5.1 FUNCTIONAL COMPONENT DEPLOYMENT

The idea of this project is adapting an existing architecture of the RingBus based Motion Controller to our motion system. The RingBus based Motion Controller uses the RingBus architecture which has been already implemented on FPGA. In the preliminary investigation part, we have performed a number of investigations about the usability of the RingBus architecture for our project. We have concluded that the RingBus architecture can satisfy our requirements. Therefore, the functional component deployment of the Mirror Controller is onto the RingBus architecture which is shown in Figure 5.54.

![Diagram](image.png)

**Figure 5.54** Deployment of the Mirror Controller

In Figure 5.54, there are a number of blocks around the RingBus. In this figure, the real time data is sent to and received from the RingBus through the Local IO System. The Control Loop block is responsible for the motion control loop calculations. The parameters, SPG profile,
internal states and the target values are stored inside the Internal States&Parameters Handler. Some diagnostic facilities like data tracing and signal injection are handled by the Facilities.

The communication lines C, D, E, F, G and H in Figure 5.48 are mapped to the RingBus which is an Avalon Streaming Bus. The lines A and B are also Avalon Streaming Bus. They are the communication lines between the Dispatcher and the Mirror Controller for the real time data. The communication lines K, L, M, N, O, P, R and S are Avalon MM Bus and directly connected to the Dispatcher.

The deployments of the Local IO System, Control Loop, Internal States&Parameters Handler and the Facilities are not shown clearly in Figure 5.54. The deployment of these components will be explained in detailed in the corresponding chapters.

5.8.5.2 INTERFACE DEPLOYMENT

It is a design decision that, Avalon MM interface is used for commands and low data transfers and Avalon ST Interface is used for streaming large data transfers, especially for the sensor and actuator data. As shown in Figure 5.55, only the Local IO System has Streaming interface with the outside world. The sensor data, actuator data, injection data and tracing data use this interface. Moreover, each component of the Mirror Controller has Avalon MM interface especially for the command handling. For receiving the commands, components use slave interface and for sending the results they use master interface. Additionally, the parameters, target values and the SPG profile are delivered to the Internal States&Parameters Handler via another Avalon MM interface. Note that, the interfaces between the components are not shown in this figure. They will be explained in the following chapters in detailed.

5.9 LOCAL IO SYSTEM

This chapter discusses the Local IO System architecture. Firstly, functions of the Local IO System are introduced. Then, decomposition of the Local IO System is explained. After the decomposition, behaviors of the Local IO System components are described. Finally, realization and deployment of the Local IO System is discussed.
5.9.1 FUNCTIONAL SPECIFICATION

The Local IO System is responsible for the streaming data coming in and out the Mirror Controller. The data that come into the Mirror Controller through the Local IO System are sensor data and injection data. The data that come out the Mirror Controller through the Local IO System are actuator data and tracing data. As shown in Figure 5.56, the Local IO System interacts with the Dispatcher, Control Loop and Facilities. The functions of the Local IO System are the followings:

- The Local IO System receives the sensor data from the Dispatcher.
- It sends the sensor data to the Control Loop.
- It receives the injection data from the Dispatcher.
- It sends the injection data to the Facilities.
- It receives the actuator data from the Control Loop.
- It sends the actuator data to the Dispatcher.
- It receives the tracing data from the Facilities.
- It sends the tracing data to the Dispatcher.
- It receives and executes the commands from the Dispatcher.
- It sends the results to the Dispatcher.

![Figure 5.56 Local IO System Component](image)

5.9.2 DECOMPOSITION OF THE LOCAL IO SYSTEM

The decomposition of the Local IO System is shown in Figure 5.57. The Local IO System consists of the Input Data Handler and the Output Data Handler.

**Rationale behind the decomposition:**
The rationale is to decouple the input data communication than the output data communication.

The functions of the Local IO System components are mentioned below.
**Input Data Handler:** Input data handler receives the sensor data from the IO System. First, the sensor data goes into an internal buffer of the Input data handler. Then, an Input data module that is internal to the Input data handler changes the mirror IDs of the sensor data. This process is explained in the following paragraph.

In the Dispatcher, a mirror ID is assigned to each sensor data to indicate the corresponding mirror. Normally, the mirror ID has \( k \) bits such that \( 2^k > N_{\text{mirror}} \). However, one RingBus motion system executes less number of mirrors. Hence there is no need to keep the current mirror ID as \( k \) bits. The Input data handler makes a new identification according to the number of mirrors into the corresponding mirror controller. For instance, if there are 32 mirrors in the Mirror Controller, the input data handler creates new mirror IDs with 5 bits. Therefore, the bandwidth of the sensor data onto the RingBus is decreased.

Input data handler receives the sensor data, changes its mirror ID and sends the sensor data to the Control Loop. For the injection data, the same story happens. Input data handler receives the injection data, changes its mirror ID and sends the data to the Facilities.

Moreover, Input data handler has another communication with the Dispatcher for the commands. The input data handler receives the corresponding commands from the Dispatcher and executes these commands. It sends the results to the Dispatcher.

**Output Data Handler:** Output data handler receives the actuator data from the Control Loop. First, the actuator data goes into an internal buffer of the Output data handler. Then, an Output data module that is internal to the Output data handler rearranges the number of bits in the mirror IDs of the corresponding mirrors.

As it is explained above, the number of bits for the mirror ID is decreased in the Input data handler. In the output data handler, the old mirror IDs (\( k \) bits IDs) are reassigned to the actuator data and they are sent to the Dispatcher. For the tracing data, the same story happens. Output data handler receives the tracing data, changes its mirror ID and sends the data to the Dispatcher.

Moreover, Output data handler has another communication with the Dispatcher for the commands. The output data handler receives the corresponding commands from the Dispatcher and executes these commands. It sends the results to the Dispatcher if required.

The communication details of the Local IO System components are shown in Table 5.9.
An FPGA based motion controller

**Communication** | **From** | **To** | **Initiator** | **Data**
---|---|---|---|---
A | Dispatcher | Input Data Handler | Dispatcher | Sensor data Injection data
B | Output Data Handler | Dispatcher | Output Data Handler | Actuator data Tracing data
C | Input Data Handler | Control Loop | Input Data Handler | Sensor data
D | Control Loop | Output Data Handler | Control Loop | Actuator data
E | Input Data Handler | Facilities | Input Data Handler | Injection data
F | Facilities | Output Data Handler | Facilities | Tracing data
G | Dispatcher | Input Data Handler Output Data Handler | Dispatcher | Command
H | Input Data Handler Output Data Handler | Dispatcher | Input Data Handler Output Data Handler | Result

Table 5.9 Communication details in the Local IO System

### 5.9.3 BEHAVIORS OF THE LOCAL IO SYSTEM COMPONENTS

There are five basic activities in the Local IO System. The first one is receiving the sensor data from the Dispatcher and delivering this data to the Control Loop which is shown in Figure 5.58.

![Sequence diagram of receiving sensor data](image)

**Figure 5.58** Sequence diagram of receiving sensor data

The second activity of the Local IO System is receiving the actuator data from the Control Loop and sending this data to the Dispatcher as shown in Figure 5.59.

![Sequence diagram of sending actuator data](image)

**Figure 5.59** Sequence diagram of sending actuator data

Moreover, the Local IO System receives the injection data from the Dispatcher and sends this data to the Facilities as shown in Figure 5.60.
Furthermore, the Local IO System receives the trace data from the Facilities and sends this data to the Dispatcher which is shown in Figure 5.61.

Finally, the Local IO System receives the commands from the Queue which is inside the Dispatcher and executes these commands as shown in Figure 5.62. Note that, they are the commands for the Local IO System. The commands for the other components of the Mirror Controller (Control Loop, Facilities and Internal States&Parameters Handler commands) are sent by the Queue directly to the corresponding component without passing through the Local IO System.

5.9.4 REALIZATION & DEPLOYMENT OF THE LOCAL IO SYSTEM

This part discusses the interface and functional components deployment of the Local IO System. The interface deployment of the Local IO System is shown in Figure 5.63. There are two modules in the Local IO System which are input data handler and output data handler. Both modules have communication with the Dispatcher, Control Loop and Facilities.

Input data handler has a streaming interface with the Dispatcher and it receives the sensor data and injection data via this interface. Input data handler has another streaming interface with the Control Loop and it sends the sensor data via this interface. The third streaming interface is between the input data handler and the Facilities. Input data handler sends the injection data via this interface.
Output data handler has a streaming interface with the Dispatcher and it sends the actuator data and tracing data via this interface. Output data handler has another streaming interface with the Control Loop and it receives the actuator data via this interface. The last streaming interface is between the output data handler and the Facilities. Output data handler receives the tracing data via this interface.

Moreover, there are one Avalon MM Master and one Avalon MM Slave interface with the Dispatcher. These two interfaces are used by input and output data handlers. The commands from the Dispatcher are received via the Slave interface and the results are sent via the Master interface.

![Interface deployment of the Local IO System](image)

**Figure 5.63** Interface deployment of the Local IO System

### 5.10 INTERNAL STATES & PARAMETERS HANDLER

This chapter discusses the Internal States & Parameters Handler architecture. Firstly, the functions of the Internal States & Parameters Handler are introduced. Then, the decomposition of the Internal States & Parameters Handler is explained. After the decomposition, behaviors of the Internal States & Parameters Handler components are described. Finally, realization and deployment of the Internal States & Parameters Handler is discussed.

#### 5.10.1 FUNCTIONAL SPECIFICATION

The parameters and the internal states which are used during the calculations in the Control Loop are stored in the Internal States & Parameters Handler. The Internal States & Parameters Handler interacts with the Control Loop and the Dispatcher as shown in Figure 5.64. The basic functions of the Internal States & Parameters Handler can be described as follows:
During initialization, the Internal States&Parameters Handler receives the parameters and SPG profile from the Dispatcher.

At the beginning of each total period, it receives the new target values from the Dispatcher.

It stores the internal states, parameters, SPG Profile and the target values.

It receives the mirror ID from the Control Loop and provides the corresponding internal states, parameters, target values and setpoint to the Control Loop.

It receives back the internal states after the calculations of the Control Loop.

It communicates with the Dispatcher for the corresponding commands.

**5.10.2 DECOMPOSITION OF THE INTERNAL STATES&PARAMETERS HANDLER**

The decomposition of the Internal States&Parameters Handler and the communication details are shown in Figure 5.65 and Table 5.10 respectively. According to Figure 5.65, the Internal States&Parameters Handler consists of the States Handler, Parameters Handler, Storage for States and the Storage for Parameters.

**Rationale behind the decomposition:**
The rationale is to decouple the communication of the parameters than the communication of the internal states.

The States Handler receives the internal states from the storage and provides them to the Control Loop. After the calculation of the Control Loop, the States Handler receives the new internal states back and stores them in the storage. The States Handler also receives the commands from the Dispatcher and sends the results back.
The Parameters Handler receives the parameters, SPG profile and target values from the Dispatcher and stores them in the storage. It provides the corresponding parameters, setpoint and target values to the Control Loop. Moreover, it receives the commands from the Dispatcher and sends the result back.

Although the Storage for States stores only the internal states, the Storage for Parameters stores the parameters, SPG profile and the target values. The parameters and the SPG profile are stored in the Storage for parameters during initialization. The new target values are stored at the beginning of each total period.

Figure 5.65 The decomposition of the Internal States & Parameters Handler

<table>
<thead>
<tr>
<th>Communication</th>
<th>From</th>
<th>To</th>
<th>Initiator</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Storage for States</td>
<td>States Handler</td>
<td>States Handler</td>
<td>Internal States</td>
</tr>
<tr>
<td>B</td>
<td>States Handler</td>
<td>Storage for States</td>
<td>States Handler</td>
<td>Internal States</td>
</tr>
<tr>
<td>C</td>
<td>Storage for Parameters</td>
<td>Parameters Handler</td>
<td>Parameters Handler</td>
<td>Parameters Setpoint Profile</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Target Values</td>
</tr>
<tr>
<td>D</td>
<td>Parameters Handler</td>
<td>Storage for Parameters</td>
<td>Parameters Handler</td>
<td>Parameters Setpoint Profile</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Target Values</td>
</tr>
<tr>
<td>E</td>
<td>Control Loop</td>
<td>States Handler</td>
<td>Control Loop</td>
<td>Mirror ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Internal States</td>
</tr>
<tr>
<td>F</td>
<td>States Handler</td>
<td>Control Loop</td>
<td>States Handler</td>
<td>Internal States</td>
</tr>
<tr>
<td>G</td>
<td>Control Loop</td>
<td>Parameters Handler</td>
<td>Control Loop</td>
<td>Mirror ID</td>
</tr>
<tr>
<td>H</td>
<td>Parameters Handler</td>
<td>Control Loop</td>
<td>Parameters Handler</td>
<td>Parameters Setpoint Profile</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Target Values</td>
</tr>
<tr>
<td>I</td>
<td>Dispatcher</td>
<td>States Handler Parameters Handler</td>
<td>Dispatcher</td>
<td>Command</td>
</tr>
<tr>
<td>J</td>
<td>States Handler Parameters Handler</td>
<td>Dispatcher</td>
<td>States Handler Parameters Handler</td>
<td>Result</td>
</tr>
<tr>
<td>K</td>
<td>Dispatcher</td>
<td>Parameters Handler</td>
<td>Dispatcher</td>
<td>Parameters Setpoint Profile</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Target Values</td>
</tr>
</tbody>
</table>

Table 5.10 Communication details of the Internal States&Parameters Handler
5.10.3 BEHAVIORS OF THE INTERNAL STATES&PARAMETERS HANDLER COMPONENTS

The parameters, SPG profile and the target values should be available inside the Internal States&Parameters Handler while the Control Loop performs its calculations. Therefore, during initialization, all parameters and the SPG profile should be stored in the Storage for the Parameters as shown in Figure 5.66. The parameters and the SPG profile are sent from the Queue to the Parameters Handler which stores them in the Storage for Parameters. Moreover, at the beginning of each total period, the target values are stored into the Storage for Parameters in a similar way.

![Figure 5.66](image)

Sequence diagram of parameters and setpoint storage in the Storage for Parameters

After initialization, the main task of the States Handler inside the Internal States&Parameters Handler is providing the internal states to the Control Loop when they are required. In the same way, the Parameters Handler delivers the parameters, setpoint and target values to the Control Loop. As shown in Figure 5.67, the Control Loop sends the mirror ID to the States Handler and Parameters Handler and receives the corresponding internal states, parameters, setpoint and target values back. Then, the Control Loop starts its execution. After the calculation of the Control Loop, the new Internal States are sent back to the States Handler.

![Figure 5.67](image)

Sequence diagram of providing the internal states and parameters to the Control Loop

The command handling of the Internal States&Parameters Handler is twofold. A command may belong to the States Handler or Parameters Handler. Moreover, a command may request an internal state or a parameter. If an internal state is requested, the States Handler accesses the Storage for States and sends the corresponding internal state to the Dispatcher. If a parameter is requested, the Parameters Handler accesses the Storage for Parameters and sends the corresponding parameter to the Dispatcher as shown in Figure 5.68.
5.10.4 REALIZATION & DEPLOYMENT OF THE INTERNAL STATES & PARAMETERS HANDLER

In this part, firstly the functional component deployment is introduced. Then, the interface deployment is explained.

5.10.4.1 FUNCTIONAL COMPONENT DEPLOYMENT

For the functional component deployment, the Internal States&Parameters Handler should be mapped to a memory block which will store the internal states and parameters. There are three types of storage: internal memory, local memory and external memory.

Internal memory is the memory internal to the FPGA. In the preliminary investigation part, it is explained that during the implementation phase Altera Stratix IV GX FPGA 530K 1517HBGA will be used. For this FPGA, there are 27,376 Kbit of embedded memory operating up to 600 MHz.

Local memory is outside the FPGA but onto the AMC board. We will use S4-AMC board which has the FPGA device Stratix IV GX 530K 1517HBGA. Onto this board, there are 128 MB DDR2 SDRAM, 512 MB DDR3 SDRAM and 128 MB Flash memories.

External memory is outside the AMC board. It can be any memory device which has suitable input and output ports to communicate with the AMC board. The size of the external memory can be very large.

The advantages and the disadvantages of using these three memory types are the following. Using internal memory is very fast, however the size of the internal memory is very small. Using the local memory is slower, but the size is significantly larger than the internal memory. There is no need to use the external memory, because the external memory is the slowest one and the size of the local memory is enough for the internal states and parameters.

In the preliminary investigation part, it is explained that the RingBus idea will be used for our project. And the processing unit of our project will be the SPU. In the SPU, there is a parameter memory and all the parameters are stored there. Moreover, there are a number of registers such that the internal states are stored there. If so, the ALU inside the SPU can access any parameter and any internal state within one clock cycle. This is the crucial idea of the SPU and

Figure 5.68 Sequence diagram of command handling in the Internal States&Parameters Handler
the scheduling in the preliminary investigation part has been done according to this idea. The important point is that, both the parameter memory and registers of the SPU should be internal memory.

However, in the preliminary investigation part, it was concluded that the internal memory of one FPGA is not enough to store all the parameters and the internal states. There are two options to solve this problem. First one is using more FPGAs: according to the investigations two or maximum three FPGAs are enough. The second idea is using both internal memory and local memory. According to this idea, the internal states will be stored in the internal memory. The parameters will be initially stored in the local memory and a set of parameters which is used in the next iteration of the motion control loop will be transferred to the internal memory. By this way, the parameters which are necessary for the running iteration of the motion control loop will be always ready in the internal memory. Afterwards, the next parameter set will be overwritten onto the previous parameter set.

Although the second idea solves the problem with one FPGA, it is very difficult to implement this solution, because an error-free synchronization is required. Moreover, for the current solution, using multiple FPGAs is acceptable by ASML because of its simplicity. Because the FPGA technology accelerates rapidly, the next generation FPGAs will have bigger internal memory size and can easily solve this problem. Therefore, for the current solution, the internal states and parameters will be stored only inside the internal memory and multiple FPGAs will be used.

5.10.4.2 INTERFACE DEPLOYMENT

This part discusses the interface deployment of the Internal States&Parameters Handler. The interface deployment of the Internal States&Parameters Handler is shown in Figure 5.69. There are two modules in the Internal States&Parameters Handler which are states handler and parameters handler. Both modules have a storage and communications with the Control Loop and Dispatcher.

States handler has one MM Master interface and one MM Slave interface with the Control Loop. The internal states are read via the Master interface and written back via the Slave interface by the Control Loop.

Parameters handler has one MM Master interface and one MM Slave interface with the Control Loop. The mirror ID is sent via the Slave interface and the parameters, setpoint and target values are read via the Master interface by the Control Loop.

Both states handler and parameters handler has one MM Master interface and one MM Slave interface with the Dispatcher. The commands are sent via the Slave interface and the results are received back via the Master interface by the Dispatcher.

Moreover, parameters handler has another MM Slave interface with the Dispatcher. Via this interface, the parameters and SPG profile are sent to the parameters handler during initialization. During runtime, the target values are also sent to the parameters handler via this interface.
5.11 CONTROL LOOP

This chapter discusses the Control Loop architecture. Firstly, the functions of the Control Loop are introduced. Then, the decomposition of the Control Loop is explained. After the decomposition, behaviors of the Control Loop components are described. Finally, realization and deployment of the Control Loop is discussed.

5.11.1 FUNCTIONAL SPECIFICATION

The calculations in the Mirror Controller are performed by the Control Loop. The Control Loop interacts with the Local IO System, Internal States&Parameters Handler, Facilities and the Dispatcher as shown in Figure 5.70. The basic functions of the Control Loop can be described as following:

- The Control Loop receives the sensor data from the Local IO System, performs the motion control loop calculations and sends the actuator data to the Local IO System.
- It receives the internal states and the parameters from the Internal States&Parameters Handler.
- It sends back the internal states to the Internal States&Parameters Handler.
An FPGA based motion controller

- It sends the finished mirror ID to the Signal Injector inside the Facilities.
- It sends the trace data to the Facilities.
- It receives the injection data from the Facilities.
- If an error or event occurs, it sends this error or event to the Error Logger or the Event Handler inside the Facilities.
- It communicates with the Dispatcher for the corresponding commands.

In Chapter 4, it is investigated that more than one mirror will be mapped to one SPU. It is a design decision that the mirrors will be executed sequentially (one by one) but the calculations inside one mirror controller will be executed in a streaming manner. The clock-cycle level pipeline will be used such that each clock cycle a new computation will start inside the SPU. By this way, the utilization of the SPU will maximize and it will never stop.

5.11.2 DECOMPOSITION OF THE CONTROL LOOP

For the decomposition of the Control Loop, there are two different ideas. First one is choosing the components of the Control Loop same as the Process Control components in the CARM architecture. Another option is creating one big component which has all the calculations of all components in the Control Loop. The advantage of the second idea is that the parallelization and the pipelining between the calculations can be exploited more efficiently if all calculations are in one component. The advantage of the first idea is that it is easy to add or remove a component to the Control Loop. Because ASML is working on this project currently, some changes may occur in the Control Loop and the second idea complicates the changes. Moreover, ASML wants to create a library to reuse the blocks in the old projects. The first idea also gives the opportunity which is the reusability of the Process Control components in the Control Loop. Therefore, the first idea will be used for the decomposition of the Control Loop.

The decomposition of the Control Loop is shown in Figure 5.71. In the Control Loop, each component has communication with its successor and predecessor component. The algorithm data is sent through this communication. Moreover, each component has a communication with the Facilities and the Internal States&Parameters Handler. Additionally, Block 10 and Block 1 have communication with the Local IO System and the Block 3 has a communication with the Dispatcher. The details about these communications are illustrated in Table 5.11.

![The decomposition of the Control Loop](image-url)
5.11.3 BEHAVIORS OF THE CONTROL LOOP COMPONENTS

The sequence diagram of calculations in the Control Loop is shown in Figure 5.72. In the sequence diagram, the Queue sends the target values to Block 3 and the Local IO System sends the sensor data to Block 1. Afterwards, each component does its calculations and sends the output to the next component. During the calculation, each component receives the corresponding parameters and internal states (if required) from Internal States & Parameters Handler and sends the internal states back. This part is not shown in Figure 5.72. The component based details during real time calculation are illustrated in Figure 5.73. In Figure 5.72, it is seen that Block 9 sends the actuator data to the Local IO System. Moreover, Block 9 sends the mirror ID to the Facilities, more precisely to the Signal Injector to inform the Signal Injector about finishing the execution of this mirror. It eliminates the risk that the injection data may be injected during the mirror calculation.
It is observable from Figure 5.72 that there are two synchronization points in the Control Loop. The first one is Block 5 and the second synchronization point is Block 8. The synchronization in these components is achieved by waiting both inputs to be ready.

In Figure 5.73, each component receives the algorithm data from the previous component and receives the parameters from the Internal States&Parameters Handler. If the component is Block 3, Block 4 or Block 6, it also takes the internal states. Afterwards, the calculation is performed with the input data, parameters and the internal states. After the calculation, the new internal states are written back to the Internal States&Parameters Handler and algorithm data is sent to the next component. Moreover, in the Control Loop each component has a number of tracing points. Therefore, each component sends the trace data to the Data Tracer if it is requested with the tracing command.

During the calculation, if some error or event occurs, the error is delivered to the Error Logger and the event is delivered to the Event handler in the Facilities as shown in Figure 5.74.
5.11.4 REALIZATION & DEPLOYMENT OF THE CONTROL LOOP

In this part, firstly the functional component deployment is introduced. Then, the interface deployment is explained.

5.11.4.1 FUNCTIONAL COMPONENT DEPLOYMENT

In sub-section 5.8.5.1, the functional component deployment of the Mirror Controller is shown. In that sub-section, the Control Loop is mapped to the SPU. Figure 5.75 shows the functional component deployment of the Control Loop.

In Figure 5.71, the Control Loop blocks are shown. There are a number of calculations in each block. These calculations are represented by ALU instructions inside the Instruction Memory of the SPU. Therefore, the Control Loop blocks will be implemented by the sets of ALU instructions inside the SPU.
5.11.4.2 INTERFACE DEPLOYMENT

In this sub-section, the interface deployment of the Control Loop blocks is mentioned. In the first paragraph, the interface between two sequential blocks is explained. Then, the interface between the Control Loop blocks and the Internal States&Parameters Handler is introduced. Afterwards, the interface of the Control Loop blocks with the Facilities is illustrated. Moreover, the first and the last block of the Control Loop communicate with the Local IO System. The interface deployment of these blocks is explained separately. Finally, the Control Loop is assumed as one big block and the interface deployment of this block is described.

According to Figure 5.71, each block in the Control Loop communicates with the previous and the next block. Because we have deployed all blocks inside one SPU, the interface between two sequential blocks is defined by the SPU structure. In Figure 5.76, MM A stands for memory mapped active interface (which will be explained later) and in Figure 5.77 MM P stands for memory mapped passive interface. Since, there is not a generic interface between two sequential blocks. According to Figure 5.75, the calculations of each block are performed by the ALU inside the SPU. The output of the block is written to the Temp Values Register through an Avalon MM bus as shown in Figure 5.77. In this communication, the block is active and the register is passive. That means the block uses the master interface and the register uses the slave interface. Afterwards, the next block starts its calculations by reading the output of the previous block from the register. In this communication, the register is slave and the block is master again. This communication is shown in Figure 5.77. Because this is not a generic interface, the representation “MM A” is used for this interface.

Each block in the Control Loop has communication with the Internal States&Parameters Handler. In Figure 5.75, the parameters are stored inside the Parameter Memory of the SPU. Therefore, the interfaces between the Control Loop and Internal States&Parameters Handler are also defined by the SPU structure. Each block sends the mirror ID to the Internal States&Parameters Handler via an Avalon MM master interface. Then, each block reads the parameters, target values and setpoints by using an Avalon MM slave interface. Block 3, Block 4 and Block 6 read the internal states via an Avalon MM slave interface and after the calculation write them back by using an Avalon MM master interface.

The ALU inside the SPU cannot execute multiple operations simultaneously. Hence, at one time only one block of the Control Loop can use the ALU. Therefore, the interface between the Control Loop and Internal States&Parameters Handler is used by only one block at a certain moment. So, this shared interface can provide the parameters to all blocks of the Control Loop in time.

Moreover, each block has communication to the Facilities with two streaming, two MM master and one MM slave interface. For data tracing and signal injection, the Control Loop uses the RingBus to send or receive data to/from Facilities. Because the RingBus is an Avalon Streaming Bus, each block has one streaming interface for data tracing and one streaming interface for signal injection. During the calculation, if an error or event occurs, it is delivered to the Error Logger or the Event handler inside the Facilities by an Avalon MM bus. For this communication, the block is active and uses Avalon MM Master interface. The Facilities is passive and uses Avalon MM Slave interface. Moreover, when the calculation of a mirror is finished, the Control Loop sends the finished mirror ID to the Facilities via another Avalon MM master interface. This is to eliminate the risk that the data is injected halfway of a mirror calculation. Furthermore, the Control Loop has one MM slave interface with the Facilities. Data tracer sends trace request command via this interface.
The interface deployment in Figure 5.76 is not valid for Block 1, Block 3 and Block 9. Since, they have access to the outside of the Control Loop. Block 3 interacts with the Queue. Block 1 and Block 9 interact with the Local IO System. The interface deployment of these components will be investigated separately.

**Interface Deployment of Block 3:** It is shown in Figure 5.78. Before Block 3, there is not a previous component. Block 3 receives its input from the Dispatcher. Therefore, one Avalon MM Slave interface is used for the communication with the Dispatcher. Other interfaces are same as in Figure 5.76.

**Interface Deployment of Block 1:** It is shown in Figure 5.79. Before Block 1, there is not a previous component. Block 1 receives its input from the Local IO System via the RingBus. Therefore, one Avalon ST interface is used for the communication with the Local IO System. Other interfaces are same as in Figure 5.76.
Interface Deployment of Block 9: It is shown in Figure 5.80. After Block 9, there is not a next component. Block 9 delivers its output to the Local IO System via the RingBus. Therefore, one Avalon ST interface is used for the communication with the Local IO System. Other interfaces are same as in Figure 5.76.

Interface Deployment of the Control Loop as one Big Block: The interface deployment of the Control Loop blocks are explained separately in the previous sections. Some of these interfaces are shared by more than one block. For instance, there is a memory mapped slave interface between each block and the Internal States&Parameters Handler to read the parameters. Indeed, there is only one interface between the Control Loop and the Internal States&Parameters Handler to send the parameters to the Control Loop blocks. Each block uses the same interface to read the parameters.

In this sub-section, the interface deployment of the Control Loop is described by assuming the Control Loop as one big block. Therefore, the interfaces between the Control Loop and Facilities, Local IO System, Dispatcher, Internal States&Parameters Handler are illustrated clearly.

The interface deployment of the Control Loop is shown in Figure 5.81. The Control Loop has two streaming interfaces with the Local IO System. One of them is for the input data and the other one is for the output data.

There are two MM master interfaces and two MM slave interfaces between the Control Loop and the Internal States&Parameters Handler. In sub-section 5.11.4.2, these interfaces are explained. The Control Loop sends the mirror ID to the Internal States&Parameters Handler via an Avalon MM master interface. Then, the Control Loop reads the parameters, target values and setpoints by using an Avalon MM slave interface. The Control Loop reads the internal states via an Avalon MM slave interface and after the calculation writes them back by using an Avalon MM master interface.
Moreover, the Control Loop has communication to the Facilities with two streaming, two MM master and one MM slave interface. These interfaces are also explained in the sub-section 5.11.4.2. The Control Loop has one streaming interface for data tracing and one streaming interface for signal injection. During the calculation, if an error or event occurs, it is delivered to the Error Logger or the Event handler inside the Facilities via an MM master interface. Moreover, when the calculation of a mirror is finished, the Control Loop sends the finished mirror ID to the Facilities via another Avalon MM master interface. Furthermore, the Control Loop has one MM slave interface with the Facilities. Data tracer sends trace request command via this interface.

Finally, the Control Loop has one MM master interface and one MM slave interface with the Dispatcher. The commands are received via the slave interface and the results are sent to the Dispatcher via the master interface.

5.12 FACILITIES

This chapter discusses the Facilities architecture. Firstly, the functions of the Facilities are introduced. Then, the decomposition of the Facilities is explained. After the decomposition, behaviors of the Facilities components are described. Finally, realization and deployment of the Facilities is discussed.

5.12.1 FUNCTIONAL SPECIFICATION

The Facilities component is a group of functions required in the CARM architecture. Data tracing, signal injection, error logging and event handling are supported by this component. The Facilities interacts with the Local IO System, Control Loop and the Dispatcher as shown in Figure 5.82. The basic functions of the Facilities are following:
The Facilities receives the injection or tracing command from the Dispatcher. It receives the injection data from the Local IO System. It sends the injection data to the Control Loop. It receives the tracing data from the Control Loop. It sends the tracing data to the Local IO System. It logs the errors generated in the Control Loop. It handles the events generated in the Control Loop. It communicates with the Dispatcher for the corresponding commands.

5.12.2 DECOMPOSITION OF THE FACILITIES

The decomposition of the Facilities and the communication details are shown in Figure 5.83 and Table 5.12 respectively. The Facilities consists of the Data Tracer, Signal Injector, Error Logger and Event Handler. Their functions are mentioned briefly below.

Rationale behind the decomposition:
The Facilities is decomposed into sub-components according to the main functions of it. For tracing the data, injecting the signal, logging the errors and handling the events different sub-components are created inside the Facilities.
Data Tracer:
- The Data Tracer receives and executes the trace command from the Dispatcher.
- It sends trace request to the Control Loop.
- It receives the trace data from the Control Loop.
- It sends the trace data to the Local IO System.
- It sends event request to and receives event confirm from the Event Handler.

Signal Injector:
- The Signal Injector receives and executes the injection command from the Dispatcher.
- It receives the injection data from the Local IO System
- It sends the injection data to the Control Loop.
- It sends event request to and receives event confirm from the Event Handler.

Error Logger:
- The Error Logger receives and executes the commands from the Dispatcher.
- It receives errors which are generated in the Control Loop.
- It sends the error to the Dispatcher.

Event Handler:
- The Event Handler receives and executes the commands from the Dispatcher.
- It receives events which are generated in the Control Loop.
- It receives event request from and sends event confirm to data tracer and signal injector.

<table>
<thead>
<tr>
<th>Communication</th>
<th>From</th>
<th>To</th>
<th>Initiator</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Dispatcher</td>
<td>Data Tracer</td>
<td>Dispatcher</td>
<td>Trace command</td>
</tr>
<tr>
<td>B</td>
<td>Data Tracer</td>
<td>Control Loop</td>
<td>Data Tracer</td>
<td>Trace request</td>
</tr>
<tr>
<td>C</td>
<td>Control Loop</td>
<td>Data Tracer</td>
<td>Control Loop</td>
<td>Trace data</td>
</tr>
<tr>
<td>D</td>
<td>Control Loop</td>
<td>Event Handler</td>
<td>Control Loop</td>
<td>Event</td>
</tr>
<tr>
<td>E</td>
<td>Dispatcher</td>
<td>Event Handler</td>
<td>Dispatcher</td>
<td>Command</td>
</tr>
<tr>
<td>F</td>
<td>Event Handler</td>
<td>Dispatcher</td>
<td>Event Handler</td>
<td>Result</td>
</tr>
<tr>
<td>G</td>
<td>Dispatcher</td>
<td>Signal Injector</td>
<td>Dispatcher</td>
<td>Injection command</td>
</tr>
<tr>
<td>H</td>
<td>Control Loop</td>
<td>Signal Injector</td>
<td>Control Loop</td>
<td>Mirror ID</td>
</tr>
<tr>
<td>I</td>
<td>Signal Injector</td>
<td>Control Loop</td>
<td>Signal Injector</td>
<td>Injection Data</td>
</tr>
<tr>
<td>J</td>
<td>Control Loop</td>
<td>Error Logger</td>
<td>Control Loop</td>
<td>Error</td>
</tr>
<tr>
<td>K</td>
<td>Dispatcher</td>
<td>Error Logger</td>
<td>Dispatcher</td>
<td>Command</td>
</tr>
<tr>
<td>L</td>
<td>Error Logger</td>
<td>Dispatcher</td>
<td>Error Logger</td>
<td>Result</td>
</tr>
<tr>
<td>M</td>
<td>Data Tracer</td>
<td>Event Handler</td>
<td>Data Tracer</td>
<td>Event request</td>
</tr>
<tr>
<td>N</td>
<td>Event Handler</td>
<td>Data Tracer</td>
<td>Event Handler</td>
<td>Event confirm</td>
</tr>
<tr>
<td>O</td>
<td>Signal Injector</td>
<td>Event Handler</td>
<td>Signal Injector</td>
<td>Event request</td>
</tr>
</tbody>
</table>
5.12.3 BEHAVIORS OF THE FACILITIES COMPONENTS

In this part, the behaviors of Data Tracer, Signal Injector, Error Logger and Event Handler are explained.

5.12.3.1 BEHAVIORS OF DATA TRACER

Data tracing is the collection of runtime data. Data tracing is used to get the runtime status and analyze the behavior of the Control Loop. In the Control Loop, there are a number of tracing points. After the Start Tracing Command is received, the data on these points should be delivered to the Data Tracer in each iteration. When the Stop Tracing Command is received, the Control Loop finishes sending the trace data to the Data Tracer. Between the Start Tracing Command and Stop Tracing Command, the trace data is sent without any request. The trace data is first stored in the Data Tracer and then sent to the Process Controller.

For tracing the data, two decisions can be made. First one is that the Control Loop always publishes all trace data to the RingBus and Data Tracer receives this data upon a request by the higher level software. Otherwise, Data Tracer looses the data. However, according to this idea, the bandwidth is a big problem and tracing data will fully utilize the RingBus.

The second idea is that the Control Loop normally does not publish the trace data. After the Data Tracer sends Start Tracing Command, the Control Loop starts to publish the trace data to the RingBus. The Data Tracer collects the trace data from the RingBus. When the Data Tracer sends Stop Tracing Command, the Control Loop finishes publishing the trace data. Because of the bandwidth problem of the first idea, the second idea will be used for the data tracing.

For the data tracing, two scenarios are possible. In the machines, there is a user interface to trace data on the tracing points in the Control Loop. For instance, user chooses the required tracing points in the motion control loop and presses the button. Afterwards, a command goes to the Data Tracer from the higher level software to trace data on the corresponding points. Then, Data Tracer sends a trace request and trace IDs to the Control Loop and the Control Loop publishes the corresponding data to the RingBus. This scenario is shown in Figure 5.84.
Another scenario is that, the user can choose a condition for starting the data tracing. For instance, if the PID error in the CTRL block of the Control Loop is outside the range, the user wants to start the tracing. For this case, the higher level software sends a command to the Data Tracer about the condition of the tracing and the trace IDs. Then, Data Tracer sends an event request to the Event Handler such that if the event happens (which is the event corresponding to the condition specified by the higher level software), the Event Handler sends an event confirm to the Data Tracer. Whenever this event occurs, Data Tracer knows that and sends a trace request to the Control Loop. This scenario is shown in Figure 5.85.

![Sequence Diagram of data tracing](image)

**Figure 5.85** Sequence Diagram of data tracing

### 5.12.3.2 BEHAVIORS OF SIGNAL INJECTOR

The signal injection is to inject a generated data into the servo Control Loop. This injection is to analyze the response of the Control Loop to the injection data. In the Control Loop, there are a number of injection points and the injection data should be delivered to the corresponding injection points when the injection command is received. In the current implementation, signal injection and data tracing are used concurrently. By observing the trace data, the response of the Control Loop to the injected data is investigated.

For the signal injection, two scenarios are possible. In the machines, there is a user interface to inject data on some specific points in the Control Loop. For instance, user chooses the required injection points in the motion control loops and presses the button. Afterwards, a command and the injection data go to the Signal Injector from the higher level software to inject data on the corresponding points. Before Signal Injector sends the injected data to the Control Loop, it must be sure that the corresponding Control Loop calculations have been finished.

In Figure 5.72, it is shown that when the Control Loop finishes its calculations, it sends the finished mirror ID to the Facilities, more precisely to the Signal Injector. When the Signal Injector receives the injection data, it checks that whether the finished mirror ID of the corresponding Control Loop has been received. If so, the Signal Injector sends the injected data to the Control Loop. If not, the Signal Injector waits till the finished mirror ID is received. This scenario is shown in Figure 5.86.
Another scenario is that, the user can choose a condition for starting the signal injection. For instance, if the PID error in the CTRL block of the Control Loop is outside the range, the user wants to start the injection. For this case, the higher level software sends the injection data and a command to the Signal Injector about the condition of the injection. Then, Signal Injector sends an event request to the Event Handler such that if the event happens (which is the event corresponding to the condition specified by the higher level software), the Event Handler sends an event confirm to the Signal Injector. Whenever this event occurs, Signal Injector knows that and sends injected data to the Control Loop. This scenario is shown in Figure 5.87.

5.12.3.3 BEHAVIORS OF ERROR LOGGER

The Error Logger logs the errors which are generated in the Control Loop and reports those errors to higher controlling software. This is for the diagnostics and maintenance purpose. In the Control Loop, an error can be caused by the followings: an invalid sensor data, the input of controller is outside the range, errors in actuators.

The behavior of the Error Logger is shown in Figure 5.88.
5.12.3.4 BEHAVIORS OF EVENT HANDLER

Event Handler logs the events which are generated in the Control Loop and communicates with the Data Tracer and the Signal Injector. Whenever an event occurs in the Control Loop, the Control Loop sends the event to the Event Handler. The event handler checks whether the event has a match with the event requests of the Data Tracer or Signal Injector. If so, the Event Handler sends an event confirm to the Data Tracer or Signal Injector. The behavior of the Event Handler is shown in Figure 5.89.

5.12.4 REALIZATION & DEPLOYMENT OF FACILITIES

In this part, firstly the functional component deployment is introduced. Then, the interface deployment is explained.
5.12.4.1 FUNCTIONAL COMPONENT DEPLOYMENT

The sizes of the storages are important for the functional component deployment of Data Tracer, Signal Injector, Error Logger and Event Handler. To define the size of the storages, we will perform some theoretical analysis.

For Data Tracer, it is mentioned in section 3.3 that there are m trace points in total and at least n of them should be supported at the same time at system level (n<m). It is also explained that, there are N_mirror mirrors in total and a mirror ID has k bits such that 2^k > N_mirror. So, for the protocol of tracing data “k” bits are required for the mirror ID and “a” bits are required for trace point ID such that 2^a > m. For the result ID and the number of trace values, some more bits are required. Assume for this protocol we will use X bits for the mirror ID, trace point ID, result ID and the number of trace values. Also assume that, each trace value has Y bits.

According to the requirements, n trace points should be supported at the same time at system level such that n<m. Hence, we should store n words in the Data Tracer in each iteration. In this protocol, the trace values indicate the values of the corresponding point in different iterations of the Control Loop. When data tracing starts, it will continue until the stop command is received. Therefore, in each iteration a new trace value is received. The arrival rate of the trace values is exactly the same as the frequency of the Control Loop. The removal rate of the trace values is dependent on the communication on the RingBus. If we assume that the frequency of sending the trace values from the Data Tracer to the Local IO System is freq_send_trace_data, the total size of the storage for the Data Tracer is;

Size of the storage = n * (X + Y * (freq_control_loop/freq_send_trace_data))

Therefore, the size of the memory block for the Data Tracer should be arranged according to the formula above.

The size of the storage in the Signal Injector should be chosen according to the number of injection points running in parallel on system level and the size of injection values for one injection point. For defining the size of the storage for Signal Injector, the way which is used for Data Tracer can be applied.

For the Error Logger, error data should contain the error ID, error content and mirror ID. The bandwidth of an error data can be defined. The number of error locations in the Control Loop is 3 for each mirror. According to the explanations above, generation rate and send out rate of the error data, the size of the storage in the Error Logger can be defined.

For the Event Handler, an event should contain the event ID, event content and mirror ID. The bandwidth of an event can be defined. According to the size of the event, number of event locations in the Control Loop, generation rate and send out rate of the event the size of the storage in the Event Handler can be defined.

5.12.4.2 INTERFACE DEPLOYMENT

This part discusses the interface deployment of the Facilities. The interface deployment of the Facilities is shown in Figure 5.90. There are four modules in the Facilities which are Event Handler, Data Tracer, Signal Injector and Error Logger.

The Event Handler has communications with the Control Loop, Dispatcher, Data Tracer and Signal Injector. The Event Handler has one MM Slave interface with the Control Loop for receiving the events generated in the Control Loop. Moreover, it has one MM Master and one MM Slave interface with the Dispatcher for the corresponding commands and their results.
The Data Tracer has communications with the Control Loop, Dispatcher, Local IO System and Event Handler. The Data Tracer has one MM Slave interface with the Dispatcher for the tracing command. It has one MM Master interface with the Control Loop for the trace request command. The Data Tracer has two streaming interfaces for carrying the trace data. One of them is between the Data Tracer and the Control Loop and the other one is between Data Tracer and the Local IO System.

The Signal Injector has communications with the Control Loop, Dispatcher, Local IO System and Event Handler. The Signal Injector has one MM Slave interface with the Dispatcher for the injection command. It has one MM Slave interface with the Control Loop for receiving the finished mirror ID. The Signal Injector has two streaming interfaces for carrying the injection data. One of them is between the Signal Injector and the Control Loop and the other one is between the Signal Injector and the Local IO System.

The Error Logger has communications with the Control Loop and Dispatcher. The Error Logger has one MM Slave interface with the Control Loop for receiving the errors generated in the Control Loop. Moreover, it has one MM Master and one MM Slave interface with the Dispatcher for the corresponding commands and their results.

![Figure 5.90 Interface deployment of the Facilities](image-url)
6 IMPLEMENTATION & REALIZATION

In the previous chapters, the architecture of the Motion System is designed to satisfy the given constraints in Chapter 3. In this part, first the implementation procedure is explained. Afterwards, the potential modifications which are required for the implementation are investigated. Finally, the working mechanism of the system is discussed.

6.1 IMPLEMENTATION PROCEDURE

After the design part which has been completed in Chapter 5, we will continue with the implementation. For the implementation phase, the following procedure will be followed. Figure 6.1 shows the procedure.

![Diagram of the implementation procedure]

**Figure 6.1** Procedure for the implementation
At the beginning, one motion control loop will be implemented on one SPU. To handle this, one SPU will be created inside the FPGA and all parameters and target values will be stored inside the FPGA at design time. During runtime, the FPGA will receive the sensor data of one mirror, perform the motion control loop calculations and send the actuator data. If this phase will be achieved successfully, we will proceed to the second phase.

In the second phase, multiple motion control loops will be implemented on one SPU. The parameters and the target values will still be stored inside the FPGA at design time. One SPU will be created inside the FPGA. The SPU will receive the sensor data of the mirrors one by one, perform the motion control loop calculations and send the actuator data to the actuators.

In the third step, multiple SPUs will be used each of which executes multiple motion control loops. The parameters and the target values will still be stored inside the FPGA at design time. Multiple SPUs will be created inside the FPGA around the same RingBus. Each SPU will receive the sensor data of the mirrors one by one, perform the motion control loop calculations and send the actuator data to the actuators.

Up to this point, the parameters and target values will be stored inside the FPGA at design time. Hence, there will not be a communication with the Process Controller in runtime. In this step, the communication with the Process Controller will also be taken into account. The FPGA takes the parameters, target values and SPG profile during runtime.

In the next step, the required number of SPUs will be created and each SPU will execute the required number of motion control loops. These numbers have been calculated in Chapter 4. To satisfy this, multiple FPGAs should be used. After this step, all motion control loops will be executed on FPGAs.

As the final step, data tracing and signal injection facilities will be added to the implementation.

### 6.2 POTENTIAL MODIFICATIONS

For this project, the RingBus Motion System is used for the communication and the SPU is used as a processing unit. However, they are created for the RingBus based Motion Controller and some modifications are required to adapt them to our project. In this part, we will explain the modifications of the SPU and the RingBus.

The SPU of the RingBus based Motion Controller is explained in sub-section 4.1.1. In the SPU, Instruction Memory is used to store the instructions and the parameters are stored in the Parameter Memory. Moreover, according to the result of the investigations in Chapter 4, the SPU will execute multiple motion control loops sequentially. Because each motion control loop uses different parameter set, the instruction set of each motion control loop is also different. Therefore, the Instruction Memory and Parameter Memory of the SPU in our project will have different sections for each motion control loop running on this SPU. Because of that, the SPU will execute each motion control loop with the instructions in the corresponding section of the Instruction Memory and with the parameters in the corresponding section of the Parameter Memory.

According to the idea in the above paragraph, when the SPU receives the sensor data, it should know which section of the Instruction Memory and Parameter Memory is used for this sensor data. This information can be carried by the RingBus. In sub-section 4.1.2, the communication on the RingBus is explained. The RingBus has 32 bits for the data and 8 bits for the channel information. The channel bits will be rearranged according to the number of SPUs around one RingBus. Moreover, some extra bits (an opcode) will be used to indicate the section of the Instruction Memory and Parameter Memory as shown in Figure 6.2.
For instance, assume that 4 SPUs are located around the same RingBus. There are 40 motion control loops and each SPU executes 10 motion control loops sequentially. The first mirror data goes to the first SPU, the second mirror data goes to the second SPU and so on. The fifth mirror data goes to the first SPU again. We will illustrate the channel bits and opcode by this scenario.

For this scenario, there are 4 SPUs, one IO System, one tracing and one injection block around the RingBus. Because there are totally 7 blocks around the RingBus and two channels are required for one block (one channel for input and one channel for output), 4 bits are required for the channel information. Because each SPU executes 10 motion control loops, each Instruction Memory should have 10 sections. Therefore, 4 bits are required for the opcode which indicates the corresponding section of the Instruction Memory.

In this scenario, the first SPU executes the mirrors 1, 5, 9, 13, 17, 21, 25, 29, 33 and 37. The second SPU executes the mirrors 2, 6, 10, 14, 18, 22, 26, 30, 34 and 38. The third SPU executes the mirrors 3, 7, 11, 15, 19, 23, 27, 31, 35 and 39. The last SPU executes the mirrors 4, 8, 12, 16, 20, 24, 28, 32, 36 and 40. The first mirror uses the first section of the Instruction Memory of the first SPU, the fifth mirror uses the second section of the Instruction Memory of the first SPU and so on. According to this information, Table 6.1 shows the data words of the sensor data of these 40 mirrors.

<table>
<thead>
<tr>
<th>Mirror</th>
<th>SPU</th>
<th>Section of the Inst. Memory</th>
<th>Opcode</th>
<th>Data</th>
<th>Input Channel</th>
<th>Output Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0000</td>
<td>Sensor data 1</td>
<td>0000</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0000</td>
<td>Sensor data 2</td>
<td>0001</td>
<td>1001</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>0000</td>
<td>Sensor data 3</td>
<td>0010</td>
<td>1010</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1</td>
<td>0000</td>
<td>Sensor data 4</td>
<td>0011</td>
<td>1011</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>0001</td>
<td>Sensor data 5</td>
<td>0000</td>
<td>1000</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>0001</td>
<td>Sensor data 6</td>
<td>0001</td>
<td>1001</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>2</td>
<td>0001</td>
<td>Sensor data 7</td>
<td>0010</td>
<td>1010</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>2</td>
<td>0001</td>
<td>Sensor data 8</td>
<td>0011</td>
<td>1011</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>3</td>
<td>0010</td>
<td>Sensor data 9</td>
<td>0000</td>
<td>1000</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>3</td>
<td>0010</td>
<td>Sensor data 10</td>
<td>0001</td>
<td>1001</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>3</td>
<td>0010</td>
<td>Sensor data 11</td>
<td>0010</td>
<td>1010</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>3</td>
<td>0010</td>
<td>Sensor data 12</td>
<td>0011</td>
<td>1011</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>4</td>
<td>0011</td>
<td>Sensor data 13</td>
<td>0000</td>
<td>1000</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>4</td>
<td>0011</td>
<td>Sensor data 14</td>
<td>0001</td>
<td>1001</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>4</td>
<td>0011</td>
<td>Sensor data 15</td>
<td>0010</td>
<td>1010</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>4</td>
<td>0011</td>
<td>Sensor data 16</td>
<td>0011</td>
<td>1011</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>5</td>
<td>0100</td>
<td>Sensor data 17</td>
<td>0000</td>
<td>1000</td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>5</td>
<td>0100</td>
<td>Sensor data 18</td>
<td>0001</td>
<td>1001</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
<td>5</td>
<td>0100</td>
<td>Sensor data 19</td>
<td>0010</td>
<td>1010</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>5</td>
<td>0100</td>
<td>Sensor data 20</td>
<td>0011</td>
<td>1011</td>
</tr>
</tbody>
</table>
## 6.3 WORKING MECHANISM

In this part, the working mechanism of the system is explained according to the given scenario in section 6.2.

According to the design in Chapter 5, the Local IO System receives the sensor data from the Dispatcher. The sensor data includes the data and the mirror ID. For the example scenario in section 6.2, 6 bits mirror ID is used to indicate 40 different mirrors. In the Local IO System, opcode and input channel bits are generated according to this mirror ID and the mirror ID is deleted. After the Local IO System, the data word is in the format of “opcode-sensor data-input channel” which is shown in Table 6.1.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Sensor data</th>
<th>Opcode</th>
<th>Input Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>1</td>
<td>6</td>
<td>0101</td>
<td>Sensor data 21</td>
<td>0000</td>
<td>1000</td>
</tr>
<tr>
<td>22</td>
<td>2</td>
<td>6</td>
<td>0101</td>
<td>Sensor data 22</td>
<td>0001</td>
<td>1001</td>
</tr>
<tr>
<td>23</td>
<td>3</td>
<td>6</td>
<td>0101</td>
<td>Sensor data 23</td>
<td>0010</td>
<td>1010</td>
</tr>
<tr>
<td>24</td>
<td>4</td>
<td>6</td>
<td>0101</td>
<td>Sensor data 24</td>
<td>0011</td>
<td>1011</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>7</td>
<td>0110</td>
<td>Sensor data 25</td>
<td>0000</td>
<td>1000</td>
</tr>
<tr>
<td>26</td>
<td>2</td>
<td>7</td>
<td>0110</td>
<td>Sensor data 26</td>
<td>0001</td>
<td>1001</td>
</tr>
<tr>
<td>27</td>
<td>3</td>
<td>7</td>
<td>0110</td>
<td>Sensor data 27</td>
<td>0010</td>
<td>1010</td>
</tr>
<tr>
<td>28</td>
<td>4</td>
<td>7</td>
<td>0110</td>
<td>Sensor data 28</td>
<td>0011</td>
<td>1011</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
<td>8</td>
<td>0111</td>
<td>Sensor data 29</td>
<td>0000</td>
<td>1000</td>
</tr>
<tr>
<td>30</td>
<td>2</td>
<td>8</td>
<td>0111</td>
<td>Sensor data 30</td>
<td>0001</td>
<td>1001</td>
</tr>
<tr>
<td>31</td>
<td>3</td>
<td>8</td>
<td>0111</td>
<td>Sensor data 31</td>
<td>0010</td>
<td>1010</td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>8</td>
<td>0111</td>
<td>Sensor data 32</td>
<td>0011</td>
<td>1011</td>
</tr>
<tr>
<td>33</td>
<td>1</td>
<td>9</td>
<td>1000</td>
<td>Sensor data 33</td>
<td>0000</td>
<td>1000</td>
</tr>
<tr>
<td>34</td>
<td>2</td>
<td>9</td>
<td>1000</td>
<td>Sensor data 34</td>
<td>0001</td>
<td>1001</td>
</tr>
<tr>
<td>35</td>
<td>3</td>
<td>9</td>
<td>1000</td>
<td>Sensor data 35</td>
<td>0010</td>
<td>1010</td>
</tr>
<tr>
<td>36</td>
<td>4</td>
<td>9</td>
<td>1000</td>
<td>Sensor data 36</td>
<td>0011</td>
<td>1011</td>
</tr>
<tr>
<td>37</td>
<td>1</td>
<td>10</td>
<td>1001</td>
<td>Sensor data 37</td>
<td>0000</td>
<td>1000</td>
</tr>
<tr>
<td>38</td>
<td>2</td>
<td>10</td>
<td>1001</td>
<td>Sensor data 38</td>
<td>0001</td>
<td>1001</td>
</tr>
<tr>
<td>39</td>
<td>3</td>
<td>10</td>
<td>1001</td>
<td>Sensor data 39</td>
<td>0010</td>
<td>1010</td>
</tr>
<tr>
<td>40</td>
<td>4</td>
<td>10</td>
<td>1001</td>
<td>Sensor data 40</td>
<td>0011</td>
<td>1011</td>
</tr>
</tbody>
</table>

Table 6.1 Data words of the example scenario

The Local IO System publishes the first data word to the RingBus, second data word to the RingBus and so on. In each clock, the Local IO System publishes the next data word to the RingBus. The first SPU looks at channel 0000, the second SPU looks at channel 0001, the third SPU looks at channel 0010 and the last SPU looks at channel 0011. When one of these SPUs takes the data word from the corresponding channel, it looks the opcode of this data word. According to the opcode, the SPU chooses the corresponding section of the Instruction Memory and Parameter Memory. When the SPU finishes the calculations, it publishes the output data to its output channel. The output data word should be in the format of “opcode-actuator data-output channel”. In the Local IO System, the mirror ID is regenerated according to the opcode and output channel bits. Afterwards, the opcode and output channel bits are deleted. The Local IO System sends the actuator data with the mirror ID to the Dispatcher. After the SPU publishes the output data to its output channel, it takes the new data word from the RingBus. By this way, SPUs never stop.

Because the SPU is data-triggered, the arrival rate of the sensor data to one SPU should be equal to the execution time of the sensor data on one SPU. In sub-section 4.2.1.1, we have calculated the period of one motion control loop. According to the example scenario given in Figure 4.11, the execution time of the sensor data on one SPU is 100 clock cycles. If there are totally M SPUs, MASU will send the sensor data of the first M mirrors as a chunk initially. The
IO System will dispatch this data and all SPUs will start execution. 100 clock cycles later, MASU will send the next M mirrors’ data as a chunk. By this way, when the execution of the first data word is finished, each SPU will take the second data word. Hence, SPUs will never stop working.
7 CONCLUSION

In this assignment, an FPGA-based motion controller for FlexRay, which is the new illumination technology of ASML products, is designed. In Chapter 3, the problem is stated and the requirements of the problem are introduced and analyzed. The idea of this project is to utilize an existing architecture which is called as the RingBus based Motion Controller. In Chapter 4 the feasibility of this idea is investigated. Based on this investigation, several conclusions can be drawn:

- Memory and logic elements usage are the important constraints to determine the required number of FPGAs which is enough to satisfy the requirements.
- Based on the analyses done with respect to memory and logic elements usage, it seems that memory usage is the bottleneck.
- Parameter memory is taken into account to have an idea of memory usage. It is explained in Chapter 4 that there are different mapping cases of motion control loops on SPU(s) which affect the memory usage. The memory size for parameters based on these mappings ranges between 2MBytes and 4MBytes.
- Our FPGA has an internal memory of ~3.5Mbytes as mentioned in Appendix G. However, this memory is not only used for parameters. There are two options to satisfy the memory usage requirements. One is using internal and external memory together with requiring one FPGA. Other one is using only internal memory but two-three FPGAs are required.

Based on the conclusions of the preliminary investigation and the given requirements, a streaming and design-time flexible architecture for the Motion System is designed in Chapter 5. The design follows a stepwise refinement approach. The Motion System is gradually decomposed into four levels as shown in Appendix C. In each level of the architecture, first the functional specifications of the component are introduced. Then, the decomposition of the component is explained. After the decomposition, behaviors of the sub-components are discussed. Finally, the realization and deployment of the sub-components are explained. Based on the architecture, several conclusions can be made:

- Multiple SPUs can be built inside the FPGA to perform the calculations of the motion control loops.
- Because the motion control loops are independent, SPUs can execute motion control loops in parallel.
- There are complex sequential processes in the Process Controller. It is not efficient to deploy the Process Controller into the FPGA. It fits better in a CPU because it requires a sequential environment.

In Chapter 6, the implementation procedure is explained. For the implementation, the processing unit and the communication unit of the RingBus based Motion Controller will be used. Since, the RingBus based Motion Controller has already been implemented on FPGA. However, some modifications are required to adapt these units to our project. These modifications are explained and the working mechanism of the system is described. The implementation will be done according to the procedure given in Chapter 6.

With the preliminary investigation work, it can be concluded that it is feasible to integrate some parts of the RingBus based Motion Controller to the FPGA based FlexRay motion controller. Together with the system level and detailed design, the following conclusions can be drawn:
- It is feasible to map the motion controller to an FPGA based platform.
- Two FPGA boards are enough to satisfy the given requirements.
- For higher frequencies (~40x faster), an FPGA based motion controller has advantages compared to the CPU-based motion controller:
  - Rack space decreases remarkably (~15x smaller).
  - Power consumption decreases tremendously (~30x less).
  - Cost decreases impressively (~30x cheaper).

To be more confident about the feasibility of the FPGA-based solution, we strongly recommend as a first step for ASML to complete the implementation and conduct experiments by following the approach we elaborate in Chapter 6.
APPENDIX A  <LEGEND FOR SHAPES AND SIGNS>

<table>
<thead>
<tr>
<th>Shape Description</th>
<th>Functional Meaning</th>
<th>Communication Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round or oval shape with grey color</td>
<td>It indicates the functional component</td>
<td>It represents the communication in the way of arrow.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In has another meaning for the functional specification sections, it indicates that the data is sensor data or sensor data request if it is reversed.</td>
</tr>
<tr>
<td>Multiple round or oval shapes</td>
<td>It indicates that there are multiple functional components</td>
<td>It represents the communication of the actuator data.</td>
</tr>
<tr>
<td>Round or oval shape with yellow color</td>
<td>It indicates the functional component which is focused on among the all components in the figure</td>
<td>It represents the communication of the data coming from or going to the Process Controller</td>
</tr>
<tr>
<td>Dashed round or oval shape</td>
<td>It indicates the component in one level higher which has components within this shape</td>
<td>Multiple sign indicates that there are more than one similar communication lines.</td>
</tr>
<tr>
<td>Square or rectangular shape</td>
<td>It indicates the deployment component</td>
<td>It represents the decomposition of the component.</td>
</tr>
<tr>
<td>Dark blue arrow</td>
<td>It represents the Avalon Streaming bus in the deployment sections</td>
<td>It shows the representation of several components by another component.</td>
</tr>
<tr>
<td>Orange arrow</td>
<td>It represents the Avalon Memory Mapped bus in the deployment sections</td>
<td>It represents the deployment of a functional component.</td>
</tr>
<tr>
<td>Green arrow</td>
<td>It represents the communication of parameters and internal states during the calculations</td>
<td></td>
</tr>
</tbody>
</table>
### APPENDIX B  <LEGEND FOR SOME WORDS>

<table>
<thead>
<tr>
<th>DEFINITION</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actuator data</td>
<td>It is the actuator data which is submitted to the actuators by the system.</td>
</tr>
<tr>
<td>Algorithm data</td>
<td>The data one block in the Control Loop submits to the other block is called as Algorithm data.</td>
</tr>
<tr>
<td>Avalon MM</td>
<td>Avalon Memory Mapped Interface is an address-based read/write interface typical of master-slave connections. Detailed information can be found in [7].</td>
</tr>
<tr>
<td>Avalon MM M</td>
<td>Avalon Memory Mapped Master initiates the transfer.</td>
</tr>
<tr>
<td>Avalon MM S</td>
<td>Avalon Memory Mapped Slave responds to the initiated transfer.</td>
</tr>
<tr>
<td>Avalon ST</td>
<td>Avalon Streaming Interface is an interface that supports the unidirectional flow of data, including multiplexed streams, packets and DSP data. Detailed information can be found in [7].</td>
</tr>
<tr>
<td>Command</td>
<td>It is the data coming from Process Controller. It has several types. Although it has several types, each type is handled equally by the system. That is why “Command” is used to represent all.</td>
</tr>
<tr>
<td>External memory</td>
<td>Memory external PPCA board.</td>
</tr>
<tr>
<td>f_desired</td>
<td>This is the desired frequency of the entire system.</td>
</tr>
<tr>
<td>f_mirror</td>
<td>This is the desired frequency of one mirror in the system.</td>
</tr>
<tr>
<td>Firmware</td>
<td>HDL code running on FPGA</td>
</tr>
<tr>
<td>Hardware</td>
<td>Electronic components and the board</td>
</tr>
<tr>
<td>Injection data</td>
<td>Injection data is a generated data injected to the Control Loop to analyze the response of the Control Loop.</td>
</tr>
<tr>
<td>Internal memory</td>
<td>Memory inside the FPGA or processor</td>
</tr>
<tr>
<td>Internal States</td>
<td>In the Control Loop calculations, some of the results are used in the next iterations. Therefore, they should be stored. They are called as internal states.</td>
</tr>
<tr>
<td>Iteration</td>
<td>The time between the Control Loop takes the sensor data and gives the actuator data is called as Iteration.</td>
</tr>
<tr>
<td>Local memory</td>
<td>Memory outside the FPGA or processor, but on same PPCA board.</td>
</tr>
<tr>
<td>Mirror ID</td>
<td>Mirror ID is used to indicate each mirror. Mirror ID has k bits such that $2^k &gt; N_{mirror}$</td>
</tr>
<tr>
<td>N_mirror</td>
<td>$N_{mirror}$ represents the total number of mirrors in the illumination system.</td>
</tr>
<tr>
<td>Parameters</td>
<td>Parameters are used during the motion control loop calculations. For instance, the coefficients in the motion control loop blocks are some of the parameters.</td>
</tr>
<tr>
<td>Result</td>
<td>Some commands need responses. Similarly with the command, all the results are treated equally by the system no matter what type it has. That is why “Result” is used to represent all responses.</td>
</tr>
<tr>
<td><strong>Sensor data</strong></td>
<td>It is the sensor data which is sent to the system by sensor unit. It does not include any information about the destination mirror controller.</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Sensor data with ID</strong></td>
<td>This is the sensor data with ID attached on it. Destination information can be found based on the ID.</td>
</tr>
<tr>
<td><strong>Setpoints</strong></td>
<td>There are a number of points on the SPG profile which are called as setpoints.</td>
</tr>
<tr>
<td><strong>Software</strong></td>
<td>Code running on processor (hardcore or softcore).</td>
</tr>
<tr>
<td><strong>SPG profile</strong></td>
<td>The SPG profile specifies the movement of the mirrors. The rotation of the mirrors is adaptable to the SPG profile.</td>
</tr>
<tr>
<td><strong>Traced data</strong></td>
<td>There are a number of trace points in the Control Loop and the data on these points are called as trace data.</td>
</tr>
</tbody>
</table>
APPENDIX C  <FUNCTIONAL OVERVIEW OF THE SYSTEM>
<table>
<thead>
<tr>
<th>ABBREVIATION</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Analog to Digital Converter</td>
</tr>
<tr>
<td>ALM</td>
<td>Adaptive Logic Module</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>ATCA</td>
<td>Advanced Telecom Computing Architecture</td>
</tr>
<tr>
<td>CARM</td>
<td>Control Architecture Reference Model</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DAC</td>
<td>Digital to Analog Converter</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>FSCR</td>
<td>FlexRay Servo Control Rack</td>
</tr>
<tr>
<td>HPPC</td>
<td>High Performance Process Controller</td>
</tr>
<tr>
<td>LAB</td>
<td>Logic Array Block</td>
</tr>
<tr>
<td>MASU</td>
<td>Mirror Angle Sensing Unit</td>
</tr>
<tr>
<td>MBLC</td>
<td>MO-Lite Based Controller</td>
</tr>
<tr>
<td>MMA</td>
<td>Micro Mirror Assembly</td>
</tr>
<tr>
<td>PPCA</td>
<td>Programmable Printed Circuit Assembly</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>ROM</td>
<td>Read Only Memory</td>
</tr>
<tr>
<td>SPU</td>
<td>Signal Processing Unit</td>
</tr>
</tbody>
</table>
8 APPENDIX E: CARM ARCHITECTURE

CARM, which is the abbreviation of “Control Architecture Reference Model”, is a reference model to create a well-defined layered model of a system. Each layer in this model has responsibilities at a specific abstraction level. CARM is used in a multidisciplinary environment covering the software, electrical and mechanical disciplines as shown in Figure 8.1.

The motivations of CARM architecture are as follows:
- It has a recognizable structure thereby easing maintenance and transferring of the model to other persons.
- It offers a view on the system that can be used as a means to let the different disciplines communicate amongst each other.
- It minimizes variation points in order to create a model that has a long lifetime.

CARM Facilities: "CARM facilities" refers to CARM implementation that has been made to make it easier for subsystem drivers to access and control the hardware. It is based on the CARM reference model. The CARM facilities are organized in the following layers:
- Application Layer
- Process Control Layer
- Transducer Layer
- Connectivity Layer

These layers can be observed in Figure 8.2.

Application: An application is responsible for a specific task within the production tools.
Process: A process is a physical quantity that is critical to handle the application responsibility.
**Transducer**: A transducer can finish the conversion from the process dimension to the control dimension.

**Connectivity**: The connectivity layer is the physical connection layer.

In the layered structure of CARM, communication can bypass layers and connectivity can be split per discipline. Moreover, one discipline can consist of multiple stacks and a layer may be omitted. These properties are shown in Figure 8.3.

Layering should result in higher layers being dependent on lower layers, but never the reverse. This means that the Transducer layer may use the Connectivity layer. The Process Control layer may use the Transducer layer. The reverse of this is not allowed.

As shown in Figure 8.4, this project will mostly focus on the Process Control layer, the Transducer layer and the Connectivity layer. Thus, we will dive into the details of these layers in the following three subsections.

### 8.1 PROCESS CONTROL LAYER

Process Control facilities provide services to control properties of physical entities like axes of stages, flows, temperatures etcetera. Feedback control systems are used to change the value of the controlled physical entities like the movement of axes or the adjustment of temperature.

Feedback control systems consist of blocks which have distinct responsibilities. These blocks are connected to each other and they periodically calculate their own outputs based on their input. The implementation of a feedback control system is distributed over of 4 components:

**Process Control Manager**: A process control manager manages resources shared by all feedback control systems. In general Process Control Manager has the following responsibilities.

- It assigns the blocks of all feedback control systems to the available HPPCs.
- It determines the computation sequence.
• It handles startup/shutdown and initialization/termination of the feedback control systems.
• It manages signal injection and tracing.

**Process Control feedback control system:** It realizes a single feedback control system. It has the following responsibilities:
• It manages the collection of parameters used by the blocks of a feedback control system.
• It provides a queue for actions for the feedback control system.
• It provides access to Blocks/Block Groups within the feedback control system.

**Process Control software:** It represents the software that is running on an HPPC. This software consists of Process Control software blocks.

**Process Control software block:** It is a component of the Process Control software. It locates inside the Process Control software component. A Process Control software block component contains the implementation of the various block types that are provided by Process Control layer. Blocks are mainly split into two parts: Worker blocks and Host blocks.

Worker blocks are the ones which are executed on HPPC. Host blocks provide a way to access the worker blocks. They are executed on a host.

**Importance of the layer for the assignment:**
For our assignment, the Process Control layer of the motion control loop architecture is very critical. In the current architecture, the Process Control software component is running on HPPCs. Other components of Process Control (Process Control Manager and Process Control feedback control system) are running on the host processor. The main functionality deployed in the Process Control software component are real time motion control loops. In other words, Process Control software is the component which handles the computational part of the motion control loops. One of the goals of our assignment is to run the motion control loops on an FPGA. That is why the Process Control layer is critical because the Process Control software component is the one which will be implemented on the FPGA.

**8.2 TRANSDUCER LAYER**

Transducer is concerned with the conversion from the process dimension to the control dimension to sense and/or steer the process value (sensors and actuators). In other words, it performs the communication between the controller and the process as shown in Figure 8.5.

![Diagram](image-url)

**Figure 8.5** Communication between the controller and the process

Transducer performs a software transducing function, which means a conversion from SI units to bits and/or vice versa. This means that the devices that write to electronics convert the client value (e.g. Newton) to values that are to be written to channels (bits). The devices that read
from electronics convert the value from the channels (bits) to the values that the client expects (e.g. 10 meters). This is illustrated in Figure 8.6.

A transducer is a device, usually electrical, electronic, or electro-mechanical, that converts one type of energy to another for the purpose of measurement or information transfer. Most transducers are either sensors or actuators. In a broader sense, a transducer is sometimes defined as any device that senses or converts a signal from one form to another. A transducer transforms a physical quantity into a quantity that can be understood using electronics (currents, tension). There are two types of transducer functions:

- It senses functions converts a physical value into a logical value.
- It actuates functions converts a logical value into a physical value.

**Importance of the layer for the assignment:**

For our assignment, the Transducer layer is a crucial part of the motion control loop architecture, because all sensors and actuators will be located in this layer. The Transducer layer lies between the Process Control and the Connectivity layers and performs the communication between the controller and the process. In the architecture, the Transducer layer is present at the host and the Transducer manager component will communicate with the Process Control software component directly or through the Process Control Manager component. The next section explains the Connectivity layer which is the communication layer.

**8.3 CONNECTIVITY LAYER**

The Connectivity layer is responsible for providing the connection. It is a physical connection layer. Figure 8.7 shows the positioning of Connectivity layer in CARM.
Figure 8.8 shows the main building blocks of Connectivity and also the context entities. The clouds above Figure 8.8 show the main functionality of the CARM Connectivity layer. These functions are described in connectivity layer functions.

![Connectivity layer functions diagram](image)

**Figure 8.8 Connectivity layer in its context**

A short description is given on each of the blocks which are located in Figure 8.8.

**Connectivity interfaces**: The interfaces of the CARM connectivity layer are defined in this component.

**Connectivity Framework**: To prevent maintenance, the connectivity framework layer does not hold technology (RIO, HSSL, ...) specific processing knowledge.

The connectivity Framework (also referred to as framework) has two main responsibilities:

- It constructs the Connectivity infrastructure at system startup (construction of the Connectivity Facilities, construction of the plug-ins). All data necessary to perform these tasks is provided by data store.
- It deals with all incoming Connectivity interface calls. The framework forwards the request to the connectivity facilities and/or Plug-in in the mesh. To forward a request to the correct destination the framework might query data store or connectivity facilities.

**Connectivity Facilities**: The Connectivity facilities are used as place holder for data which should be available for the framework and plug-ins during the lifetime of the Connectivity manager instance.

A ‘Connectivity facility’ should comply with the following general guidelines:

- Caching of data which needs to be accessed frequently within the framework or plug-in mesh. Data which needs cooking or where relations need to be resolved before it is usable for the Connectivity framework or plug-ins in the mesh.
- Memory usage reduction can be accomplished because data is needed by more than one plug-in or framework module. (single point of definition and minimize redundancy of data)
Plug-in mesh: For each board that is under control of the Connectivity layer, a plug-in exists. Each type of electronics board will typically have its own specific implementation (represented by the plug-in object) because of the technological differences compared to other boards. Each electronics board in the system will have its plug-in counter part as a software object running on the Host node. If electronic boards are connected with each other physically by a technology specific bus, the plug-in counterparts in software use a technology specific plug-in interface. A mesh of electronics boards in the physical world results in a similar plug-in mesh instantiated in the context of the Connectivity process on the Host node.

Plug-in interfaces: Two categories of plug-in interfaces can be distinguished:
- Technology independent interfaces; these are the ones that are used by the framework.
- Technology dependent interfaces; which are used by the plug-in mesh to communicate with each other.

Electronics Boards mesh: The collection of electronic boards within or attached to the physical electronics rack which is under Connectivity control.

Importance of the layer for the assignment:
The Connectivity layer is important for our assignment. Deploying the motion control loop in an FPGA depends on the used hardware architecture. According to the hardware platform, Connectivity layer may have different duties.

In addition, motion control loops always receive data from the sensors and send data to the actuators. Besides, communication is necessary between the host and the FPGA. Interfaces for all these communications must be defined beforehand. It has a direct effect on the design and its performance. Moreover, communication within the FPGA is very crucial. There are real-time requirements that we need to satisfy.
APPENDIX F: RINGBUS BASED MOTION CONTROLLER

The RingBus based Motion Controller has been designed to decrease the vibrations of the stages in ASML machines. It consists of a number of active vibration dampers. Each damper has a sensor and an actuator. For each damper, there is a corresponding control loop and each control loop works independently. The calculation primitives of the control loops in this project and the motion control loops in FlexRay project are very similar. Moreover, the diagnostic facilities like tracing and injection are also present in the RingBus based Motion Controller.

Because the RingBus based Motion Controller has already been implemented on an FPGA and because of the similarities between this project and FlexRay Project we will utilize the idea of the RingBus based Motion Controller to implement the FlexRay motion control loops to the FPGA.

The control loops in the RingBus based Motion Controller are managed by an FPGA board. The FPGA controls the amplifiers through DACs and the amplifiers control the actuators. The internal architecture of the FPGA is shown in Figure 9.1. The RingBus based Motion Controller uses the RingBus Motion System for the real-time communication. The input data from ADCs directly goes to the RingBus Motion System and the output data is sent to the DACs by the RingBus Motion System. The real time calculations for the control loops are executed in a processing unit which is called as SPU. SPUs are located around the RingBus Motion System. The RingBus Motion System and the SPU are explained in section 4.1. In our project we use the SPU as a processing unit and the RingBus for the communication.

![RingBus Based Motion Controller FPGA](image)
10 APPENDIX G: FPGA

The FPGA device onto our board is Altera Stratix IV GX FPGA 530K 1517HBGA with part number, “EP4SGX530KH40I3”. The memory blocks, DSP blocks, logic elements, etc. will be explained for our specific FPGA in the following sections. Detailed information on the FPGA can be found in [3].

![FPGA Fabric and I/O Features](image)

**Figure 10.1** Stratix IV GX Chip view

**Device Core Features:**

The FPGA device onto our board which is Altera Stratix IV GX 530K 1517HBGA has the followings:

- It has 531,200 LEs efficiently packed in 212,480 adaptive logic modules (ALMs).
- It has 10 ALMs per LAB which deliver faster performance, improved logic utilization, and optimized routing.
- It has programmable power technology including a variety of process, circuit, and architecture optimizations and innovations.
- Its programmable power technology is available to select power-driven compilation options for reduced static power consumption.

**Embedded Memory:**

To efficiently address the needs of diversified FPGA designs, TriMatrix embedded memory architecture into Stratix IV GX provides three different memory block sizes:

- 640-bit MLAB,
- 9-Kbit M9K,
- 144-Kbit M144K.

The FPGA onto our board has totally 27,376 Kbit of embedded memory operating up to 600 MHz. Each memory block is independently configurable to be a single- or dual-port RAM, FIFO, ROM, or shift register.

**DSP Blocks:**

Altera Stratix IV FPGAs have flexible DSP blocks which are configurable as 18 × 18-bit full-precision multipliers at up to 600 MHz with rounding and saturation capabilities. Moreover, these DSP blocks have faster operation speed due to fully pipelined architecture and built-in
addition, subtraction, and accumulation units to combine multiplication results. Furthermore, they are optimally designed to support advanced features such as adaptive filtering, barrel shifters and finite and infinite impulse response (FIR and IIR) filters.

<table>
<thead>
<tr>
<th>Feature</th>
<th>EP4SGX530KH40I3</th>
</tr>
</thead>
<tbody>
<tr>
<td>M9K Blocks (256x36 bits)</td>
<td>1280</td>
</tr>
<tr>
<td>M144K Blocks (2048x72 bits)</td>
<td>64</td>
</tr>
<tr>
<td>Total Memory (MLAB+M9K+M144K) Kbits</td>
<td>27376</td>
</tr>
<tr>
<td>Embedded Multipliers 18x18</td>
<td>1024</td>
</tr>
<tr>
<td>PLLs</td>
<td>8</td>
</tr>
<tr>
<td>User I/Os</td>
<td>744</td>
</tr>
</tbody>
</table>

Table 10.1  Stratis IV GX530K 1517 Device Features

The device features of the corresponding FPGA onto our board are shown in Table 10.1. Now, the memory blocks and the DSP blocks of the FPGA will be explained more detailed in the next two sections.

Memory Blocks into Stratis IV

This section describes the TriMatrix embedded memory blocks into Stratis IV GX530K 1517 device. TriMatrix embedded memory blocks provide three different sizes of embedded SRAM to efficiently address the needs of Stratis IV FPGA designs. TriMatrix memory includes 640-bit MLABs, 9-Kbit M9K blocks, and 144-Kbit M144K blocks. MLABs have been optimized to implement filter delay lines, small FIFO buffers, and shift registers. M9K blocks are used for general purpose memory applications and the M144K blocks for processor code storage, packet buffering, and video frame buffering. Each embedded memory block can be configured independently to be a single or dual-port RAM, FIFO buffer, ROM, or shift register using the Quartus II MegaWizardTM Plug-In Manager. TriMatrix memory capacity and distribution in our FPGA are shown in Table 10.2. Detailed information on the memory blocks of FPGA can be found in [4].

<table>
<thead>
<tr>
<th>Device</th>
<th>MLABs</th>
<th>M9K Blocks (Dedicated Memory Blocks Only) (Kb)</th>
<th>M144K Blocks (Including MLABs) (Kb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EP4SGX530</td>
<td>10,624</td>
<td>1280</td>
<td>64</td>
</tr>
</tbody>
</table>

Table 10.2  Trimatrix Memory Capacity and Distribution in Stratis IV GX530K 1517 Device

While the M9K and M144K memory blocks are dedicated resources, the MLABs are dual-purpose blocks. They can be configured as regular LABs. Ten ALMs make up one MLAB. Each ALM in an MLAB can be configured as either a 64x1 or 32x2 block, resulting in a 64x10 or 32x20 simple dual-port SRAM block in a single MLAB.

All TriMatrix memory blocks have built-in-parity-bit support. The ninth bit associated with each byte can store a parity bit or serve as an additional data bit.

All TriMatrix memory blocks support byte enables that mask the input data so that only specific bytes of data are written. The unwritten bytes retain the previously written values. The write enable (wren) signals, along with the byte enable (byteena) signals, control the RAM blocks' write operations.
Stratix IV M9K and M144K blocks support packed mode. The packed mode feature packs two independent single-port RAMs into one memory block.

All Stratix IV memory blocks support address clock enable, which holds the previous address value for as long as the signal is enabled (addressstall = 1). When the memory blocks are configured in dual-port mode, each port has its own independent address clock enable. The default value for the address clock enable signals is low (disabled).

M9K and M144K memory blocks support mixed data widths inherently. MLABs can support mixed data widths through emulation using the Quartus II software. When using simple dual-port, true dual-port, or FIFO modes, mixed width support allows reading and writing different data widths to a memory block.

Stratix IV M144K blocks have built-in support for ECC when in ×64-wide simple dual-port mode. ECC allows detecting and correcting data errors in the memory array. The M144K blocks have a single-error-correction double-error-detection (SECDED) implementation. SECDED can detect and fix a single bit error in a 64-bit word, or detect two bit errors in a 64-bit word. It cannot detect three or more errors.

**Memory Modes**

Stratix IV TriMatrix memory blocks allow implementing fully synchronous SRAM memory in multiple modes of operation. M9K and M144K blocks do not support asynchronous memory (unregistered inputs). MLABs support asynchronous (flow-through) read operations. The following memory modes can be used:

**Single-Port RAM Mode:** All TriMatrix memory blocks support single-port mode. Single-port mode allows doing either one-read or one-write operation at a time. Simultaneous reads and writes are not supported in single-port mode.

**Simple Dual-Port Mode:** All TriMatrix memory blocks support simple dual-port mode. Simple dual-port mode allows performing one read and one write operation to different locations at the same time. Write operation happens on port A; read operation happens on port B.

**True Dual-Port Mode:** Stratix IV M9K and M144K blocks support true dual-port mode. This mode allows performing any combination of two port operations: two reads, two writes, or one read and one write at two different clock frequencies. The widest bit configurations of the M9K and M144K blocks are 512x16-bit (or 512x18-bit with parity) and 4Kx32-bit (or 4Kx36-bit with parity) respectively.

In true dual-port mode, any memory location is accessible at any time from either port. When accessing the same memory location from both ports, the possible write conflicts must be avoided; since no conflict resolution circuitry is built into the Stratix IV TriMatrix memory blocks. A write conflict happens when writing to the same address location from both ports at the same time is attempted.

**Shift-Register Mode:** All Stratix IV memory blocks support shift register mode. Some DSP applications require local data storage, traditionally implemented with standard flip-flops that quickly exhaust many logic cells for large shift registers. A more efficient alternative is to use embedded memory as a shift-register block, which saves logic cell and routing resources. The memory blocks can be cascaded to implement larger shift registers.

**ROM Mode:** All Stratix IV TriMatrix memory blocks support ROM mode. The address lines of the ROM are registered on M9K and M144K blocks, but can be unregistered on MLABs. The ROM read operation is identical to the read operation in the single-port RAM configuration.
FIFO Mode: All TriMatrix memory blocks support FIFO mode. MLABs are ideal for designs with many small, shallow FIFO buffers. The Quartus II software FIFO MegaWizard Plug-In Manager is used to implement FIFO buffers in the design.

Clocking Modes
Stratix IV TriMatrix memory blocks support the following clocking modes:

Independent Clock Mode: Stratix IV TriMatrix memory blocks can implement independent clock mode for true dual-port memories. In this mode, a separate clock is available for each port. Clock A controls all registers on the port A side; clock B controls all registers on the port B side.

Input/Output Clock Mode: Stratix IV TriMatrix memory blocks can implement input/output clock mode for true dual-port and simple dual-port memories. In this mode, an input clock controls all registers related to the data input to the memory block including data, address, byte enables, read enables, and write enables. An output clock controls the data output registers.

Read/Write Clock Mode: Stratix IV TriMatrix memory blocks can implement read/write clock mode for simple dual-port memories. In this mode, a write clock controls the data-input, write-address, and write-enable registers. Similarly, a read clock controls the data-output, read-address, and read-enable registers.

Single Clock Mode: Stratix IV TriMatrix memory blocks can implement single-clock mode for true dual-port, simple dual-port, and single-port memories. In this mode, a single clock, together with a clock enable, is used to control all registers of the memory block.

Finally, The Quartus II software automatically partitions user-defined memory into embedded memory blocks by taking into account both speed and size constraints placed on a design. Because there is no conflict resolution circuitry into the memory blocks, conflict resolution logic must be implemented external to the memory block to avoid address conflicts.

DSP Blocks into Stratix IV
In Stratix IV devices, DSP blocks are optimized to support DSP applications (FIR filters, IIR filters, FFT functions, encoders, etc.) requiring high data throughput. The DSP blocks consist of a combination of dedicated elements that perform multiplication, addition, subtraction, accumulation, summation and dynamic shift operations. Detailed information on the DSP blocks of FPGA can be found in [5].

<table>
<thead>
<tr>
<th>Device</th>
<th>DSP Blocks</th>
<th>9x9 Multipliers</th>
<th>12x12 Multipliers</th>
<th>18x18 Multipliers</th>
<th>18x18 Complex</th>
<th>36x36 Multipliers</th>
<th>18x36 Multipliers</th>
<th>18x18 Multipliers</th>
</tr>
</thead>
<tbody>
<tr>
<td>EP4SGX530</td>
<td>128</td>
<td>1024</td>
<td>768</td>
<td>512</td>
<td>256</td>
<td>256</td>
<td>512</td>
<td>1024</td>
</tr>
</tbody>
</table>

Table 10.3 Number of DSP Blocks in Stratix IV GX530K 1517 Device

Each DSP block occupies four LABs in height and can be divided further into two half blocks that share some common clock signals as shown in Figure 10.2.
In Stratix IV devices, the fundamental building block is a pair of 18x18-bit multipliers followed by a first-stage 37-bit addition/subtraction unit as shown in Figure 10.3.

This basic building block is useful for building more complex structures, such as complex multipliers and 36x36 multipliers.

Each Stratix IV DSP block contains four two-multiplier adder units (2 two-multiplier adder units per half block). Therefore, there are eight $18 \times 18$ multiplier functionalities per DSP block.

Depending on the selected mode, one can bypass all register stages except accumulation and loopback mode. In these two modes, a set of register must be enabled. If the register is not enabled, an infinite loop occurs.

To support commonly found FIR-like structures efficiently, a major addition to the DSP block in Stratix IV devices is the ability to propagate the result of one half block to the next half block completely within the DSP block without additional soft logic overhead. This issue is shown in Figure 10.4.
Stratix IV DSP blocks can operate in different modes simultaneously. Each DSP block consists of two identical halves (the top half and bottom half). Each half has four 18x18 multipliers and is fully independent except for the sharing of three signals (clock, ena and aclr).

**Stratix IV DSP Block Resource Description**

The DSP block consists of the following elements:

- Input register bank,
- Four two-multiplier adders,
- Pipeline register bank,
- Two second-stage adders,
- Four rounding and saturation logic units,
- Second adder register and output register bank.

These elements are observed in Figure 10.5.
Figure 10.5  Half DSP Block Architecture

**Input Register Bank:** All of the DSP block registers are triggered by the positive edge of the clock signal and are cleared upon power up. Each multiplier operand can feed an input register or go directly to the multiplier, bypassing the input registers. Every DSP block has nine 18-bit data input register banks per half DSP block.

The first multiplier in every half DSP block in Stratix IV devices has a multiplexer for the first multiplier B-input register to select between general routing and loopback.

**Multiplier and First-Stage Adder:** The multiplier stage supports 9x9, 12x12, 18x18, 36x36 multipliers and 18x18 complex multiplier. Each multiplier operand can be a unique signed or unsigned number. Two dynamic signals (signa and signb) control the representation of each operand, respectively. Each half block has its own signa and signb signal.

**First Stage Adder:** The first-stage adder block has the ability to perform addition and subtraction. There are four first-stage adders in a DSP block (two adders per half DSP block). The outputs of the multipliers are the only outputs that can feed into the first-stage adder.

**Second Stage Adder:** There are four individual 44-bit second-stage adders per DSP block (two adders per half DSP block). The second stage adder can be used as an accumulator (44-bits maximum). Or this can be configured as the final stage of a 36-bit multiplier or a chained output summation.

**Rounding and Saturation Stage:** The rounding and saturation logic units are located at the output of the second-stage adder. There are two rounding and saturation logic units per half DSP block.

**Second Adder and Output Registers:** The second adder register and output register banks are two banks of 44-bit registers that can be combined to form larger 72-bit banks to support 36 × 36 output results.

Depending on the operational mode of the DSP block, the output selection unit selects whether the outputs of the DSP blocks come from the outputs of the multiplier block, first-stage adder, pipeline registers, second-stage adder, or the rounding and saturation logic unit.
Stratix IV Operational Mode Descriptions

There are different operational modes in Stratix IV devices as following:

- **Independent Multiplier Modes**
- **Two-Multiplier Adder Sum Mode**
- **High-Precision Multiplier Adder Mode**
- **Multiply Accumulate Mode**
- **Shift Modes**
- **Rounding and Saturation Mode**

**Independent Multiplier Modes:** In this mode, the DSP block performs individual multiplication operations for general-purpose multipliers. 9-, 12-, 18-, 36-Bit and Double Multiplier are in this mode.

- **9-, 12-, and 18-Bit Multiplier:** Each DSP block multiplier can be configured for 9-, 12-, or 18-bit multiplication. A single DSP block can support up to eight individual 9 × 9 multipliers, six individual 12 × 12 multipliers, or four individual 18 × 18 multipliers. For operand widths up to 9 bits, a 9 × 9 multiplier is implemented. For operand widths from 10 to 12 bits, a 12 × 12 multiplier is implemented, and for operand widths from 13 to 18 bits, an 18 × 18 multiplier is implemented.

- **36-Bit Multiplier:** 36x36 multiplier can efficiently be constructed by using four 18x18 multipliers. Stratix IV devices can have up to two 36-bit multipliers per DSP block (one 36-bit multiplier per half DSP block).

- **Double Multiplier:** It is basically a 54 x 54-bit multiplier and can be built by using basic 18 x 18 multipliers shifters, and adders.

**Two-Multiplier Adder Sum Mode:** In this mode, the DSP block can implement four 18-bit two-multiplier adders (2 two-multiplier adders per half DSP block). This mode can be utilized to take the sum or difference of two multiplier outputs. For applications such as FFTs, complex FIR and IIR filters, the two-multiplier adder mode is useful. 18 x 18 complex multiplier and four-multiplier adder can be achieved with a small modification of this mode.

- **18 x 18 Complex Multiply:** A single half DSP block can implement one 18-bit complex multiplication in two-multiplier adder mode.

- **Four Multiplier Adder:** A single half DSP block can implement one four-multiplier adder. The outputs of two of the four multipliers are initially summed in the two first-stage adder blocks. The results of these two adder blocks are then summed in the second-stage adder block to produce the final four-multiplier adder result.

**High-Precision Multiplier Adder Mode:** In this mode, the DSP block can implement 2 two-multiplier adders, with multiplier precision of 18 x 36 (one two-multiplier adder per half DSP block). This mode is useful in filtering or FFT applications where a data path greater than 18 bits is required, yet 18 bits is sufficient for the coefficient precision.

**Multiply Accumulate Mode:** In this mode, a single DSP block can implement up to two independent 44-bit accumulators. To handle this, the second-stage adder is configured as a 44-bit accumulator or subtractor. The output of the DSP block is looped back to the second-stage adder and added or subtracted with the two outputs of the first-stage adder block.
Multiply Accumulate Mode: In this mode, a single DSP block can implement up to two independent 44-bit accumulators. To handle this, the second-stage adder is configured as a 44-bit accumulator or subtractor. The output of the DSP block is looped back to the second-stage adder and added or subtracted with the two outputs of the first-stage adder block.

Shift Modes: Stratix IV devices support the following shift modes for 32-bit input only:

- Arithmetic shift left,
- Arithmetic shift right,
- Logical shift left,
- Logical shift right,
- 32-bit rotator or barrel shifter.

Two control signals, rotate and shift_right, together with the signa and signb signals, determine the shifting operation.

Rounding and Saturation Mode: In general, rounding is used to limit bit growth and its side effects; saturation is used to reduce overflow and underflow side effects. Two rounding modes are supported in Stratix IV devices:

- Round-to-nearest-integer mode,
- Round-to-nearest-even mode.

Two saturation modes are supported:

- Asymmetric saturation mode,
- Symmetric saturation mode.
APPENDIX H <FSCR>

FSCR is the current platform for the servo calculations of the FlexRay module which is illustrated in Figure 2.4. Currently, there are several active ATCA Rack configurations and FSCR is one of these configurations. FSCR is based on a standard 5 Slots ATCA Rack that is shown in Figure 2.3.

![FSCR based on 5 Slots ATCA Rack](image)

The layout of the FSCR can be divided into four entities. Important ones are explained below:

- **FSCR Base Interface (1G Ethernet):** The Base Interface has several responsibilities:
  - It does firmware uploading to HPPC’s (after powerup).
  - It uploads machine constants needed for servo application (during initialization).
  - It transfers function calls to host HPPC.
  - It downloads pupil shape.

The layout of the FSCR Base Interface is shown in Figure 2.4.

![FSCR Base Interface](image)
**FSCR Fabric Network Interface:** The Fabric Interface allocates several differential pairs per Fabric Channel. A differential pair will be called a lane. The Fabric Interface is typically used to move data between the boards and the outside network. The Fabric of the FSCR is SRIO. FSCR SRIO fabric interface is shown in Figure 2.5.

The SRIO in FSCR fabric interface is built on optical fibers. Because of the real-time data communication in FSCR fabric interface, optical fibers are required to communicate in the order of nanoseconds. Due to the required protocols of Ethernet, it cannot be used for real-time data communication.

![FSCR SRIO fabric interface](image)

**FSCR Physical Layout:** The FSCR holds 5 ATCA blades that each has 4 slots for AMC boards. In FlexRay the AMC boards are:
- SRI O to Fiber breakout board.
- board that controls timing.
- hosts that are running driver SW, PowerPC based processor boards.
- several motion controllers, PowerPC based processor boards (known as HPPC boards).
## APPENDIX I <REFERENCE DOCUMENTS>

<table>
<thead>
<tr>
<th>REFERENCE</th>
<th>TITLE</th>
</tr>
</thead>
</table>