Development of a Predictable Hardware Architecture Template and Integration into an Automated System Design Flow

MARCUS MIKULCAK

Masters’ Degree Project
Second level, 30.0 HEC
Stockholm, Sweden June 2013

TRITA-ICT-EX-2013:138
Development of a Predictable Hardware Architecture Template and Integration into an Automated System Design Flow

Master of Science Thesis in System-on-Chip Design
Stockholm, June 2013
TRITA-ICT-EX-2013:138

Author: Marcus Mikulcak
Examiner: Assoc. Prof. Ingo Sander
Supervisor: Seyed Hosein Attarzadeh Niaki
Abstract

The requirements of safety-critical real-time embedded systems pose unique challenges on their design process which cannot be fulfilled with traditional development methods. To ensure their correct timing and functionality, it has been suggested to move the design process to a higher abstraction level, which opens the possibility to utilize automated correct-by-design development flows from a functional specification of the system down to the level of Multiprocessor Systems-on-Chip. ForSyDe, an embedded system design methodology, presents a flow of this kind by basing system development on the theory of Models of Computation and side-effect-free processes, making it possible to separate the timing analysis of computation and communication of process networks. To be able to offer guarantees on the timing of tasks implemented on a Multiprocessor System-on-Chip, the hardware platform needs to provide predictability and composability in every component, which in turn requires a range of special considerations in its design. This thesis presents a predictable and composable FPGA-based Multiprocessor System-on-Chip template based on the Altera Nios II soft processor and Avalon Switch Fabric interconnection structure and its integration into the automated ForSyDe system design flow. To present the functionality and to test the validity of timing predictions, two sample applications have been developed and tested in the context of the design flow as well as on the implemented hardware platform.

Keywords: Predictability, Composability, Multiprocessor, Architecture, FPGA, Altera, ForSyDe
Contents

List of Figures v
List of Tables vii
List of Listings viii
List of Abbreviations ix

1 Introduction & Background 1
   1.1 Embedded systems .................................................. 1
       1.1.1 Real-time systems ............................................. 1
       1.1.2 Hard real-time systems ....................................... 2
       1.1.3 Soft real-time systems ....................................... 2
       1.1.4 Designing embedded systems ................................. 3
   1.2 Problem ................................................................. 5
   1.3 Method ................................................................. 6
   1.4 Outline ................................................................. 6

2 Predicting Timing Behavior 9
   2.1 The need for timing analysis ....................................... 9
   2.2 Calculation of execution time bounds ............................ 10
       2.2.1 Problems and requirements .................................... 10
       2.2.2 Existing approaches to predicting timing behavior ....... 13

3 Design of Time-Predictable Hardware Architectures 17
   3.1 Requirements for predictability .................................... 17
       3.1.1 Principles for a predictable processor ....................... 17
       3.1.2 Principles for a predictable memory controller ........... 18
       3.1.3 Principles for a predictable interconnection structure .... 19
   3.2 Definitions of the term predictability .......................... 20
       3.2.1 Predictability through determinism ......................... 20
3.2.2 Is time-predictability quantifiable? ................................. 23
3.2.3 Relation between worst-case and best-case timing behavior ................................. 24

4 Existing Predictable Architectures ................................. 27
4.1 CompSOC ................................................................. 27
4.1.1 Goals ................................................................. 27
4.1.2 Platform overview .................................................. 28
4.1.3 The hardware platform .............................................. 28
4.2 Merasa ................................................................. 33
4.2.1 Predictability ...................................................... 34
4.2.2 Composability ...................................................... 35
4.2.3 The hardware platform .............................................. 35
4.3 Comparison and summary ........................................... 40

5 System Design using ForSyDe ................................. 43
5.1 Introduction .............................................................. 43
5.1.1 Models of Computation ........................................... 44
5.1.2 Synchronous Model of Computation ........................................... 45
5.1.3 Synchronous Data Flow ........................................... 45
5.2 ForSyDe processes ....................................................... 46
5.3 Modeling framework .................................................... 47
5.3.1 ForSyDe SystemC ...................................................... 48
5.4 Design methodology ..................................................... 50
5.4.1 Initial flow components ........................................... 52
5.4.2 Design flow steps ...................................................... 53
5.5 Conclusion ............................................................... 56

6 Development & Implementation ................................. 57
6.1 Altera design tools ..................................................... 57
6.1.1 Nios II processors ...................................................... 58
6.1.2 Memory elements ...................................................... 62
6.1.3 Interconnection structure ........................................... 64
6.1.4 System creation using Qsys ........................................... 71
6.2 Creating an automated Nios II design flow ........................................... 72
6.2.1 Flow inputs ...................................................... 73
6.2.2 SOPC Builder model creation ........................................... 80
6.2.3 Arbitrator modification ........................................... 84
6.2.4 Quartus II project creation ........................................... 85
6.2.5 Nios II project creation ........................................... 86
6.2.6 Combined architecture creation flow ........................................... 87
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Top-down design process</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Two types of timing anomalies</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>The uncertainties of dynamic timing analysis</td>
<td>14</td>
</tr>
<tr>
<td>3.1</td>
<td>Lower and upper bounds in relation to real WCET and BCET values</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>The basic CompSOC architecture template</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>Contention-free TDM routing in the Æthereal NoC</td>
<td>30</td>
</tr>
<tr>
<td>4.3</td>
<td>Memory tiles in the CompSOC platform</td>
<td>31</td>
</tr>
<tr>
<td>4.4</td>
<td>Merasa platform overview</td>
<td>34</td>
</tr>
<tr>
<td>4.5</td>
<td>A single Merasa SMT core</td>
<td>36</td>
</tr>
<tr>
<td>4.6</td>
<td>Merasa scratchpad memory</td>
<td>38</td>
</tr>
<tr>
<td>5.1</td>
<td>An example ForSyDe process network</td>
<td>44</td>
</tr>
<tr>
<td>5.2</td>
<td>An example SDF process network</td>
<td>45</td>
</tr>
<tr>
<td>5.3</td>
<td>An example SDF schedule generation</td>
<td>46</td>
</tr>
<tr>
<td>5.4</td>
<td>Conversion of an SDF into an HSDF network</td>
<td>47</td>
</tr>
<tr>
<td>5.5</td>
<td>The ForSyDe process constructor approach</td>
<td>48</td>
</tr>
<tr>
<td>5.6</td>
<td>The ForSyDe design flow</td>
<td>52</td>
</tr>
<tr>
<td>5.7</td>
<td>Three example mapping solutions</td>
<td>54</td>
</tr>
<tr>
<td>6.1</td>
<td>The Nios II processor architecture</td>
<td>58</td>
</tr>
<tr>
<td>6.2</td>
<td>SRAM write operation timing</td>
<td>63</td>
</tr>
<tr>
<td>6.3</td>
<td>SRAM read operation timing</td>
<td>63</td>
</tr>
<tr>
<td>6.4</td>
<td>On-chip memory read and write operation timing</td>
<td>64</td>
</tr>
<tr>
<td>6.5</td>
<td>Typical transfer across the Avalon Switch Fabric</td>
<td>65</td>
</tr>
<tr>
<td>6.6</td>
<td>Arbitration of continuous simultaneous transfer requests</td>
<td>66</td>
</tr>
<tr>
<td>6.7</td>
<td>The multi-master arbitration mechanism in the Avalon Switch Fabric</td>
<td>67</td>
</tr>
<tr>
<td>6.8</td>
<td>Arbitration of two master ports with a gap in transfer requests</td>
<td>68</td>
</tr>
<tr>
<td>6.9</td>
<td>Examples of TDM tables of the developed arbitrator</td>
<td>68</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Different hardware approaches to minimizing execution time variation. . . . 40
6.1 Execution time of instructions on a Nios II/e core. . . . . . . . . . . . . 62
7.1 ALF language construct costs. . . . . . . . . . . . . . . . . . . . . . . 99
7.2 Tahmuras cost indicators. . . . . . . . . . . . . . . . . . . . . . . . . 100
7.3 Statically determined WCEC and BCEC of each JPEG encoder process. . 104
7.4 Deviation of WCEC and BCEC of each JPEG encoder process. . . . . . . 106
7.5 Measured and predicted WCCC values. . . . . . . . . . . . . . . . . . . . 107
7.6 Delay inferred by conflicting access to a shared slave module . . . . . . 108
7.7 Tahmuras-generated resource usage predictions . . . . . . . . . . . . . 108
7.8 BCEC and WCEC values of each SUSAN process. . . . . . . . . . . . . . 110
## List of Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>A ForSyDe signal example</td>
<td>49</td>
</tr>
<tr>
<td>5.2</td>
<td>The definition of a ForSyDe SDF function</td>
<td>49</td>
</tr>
<tr>
<td>5.3</td>
<td>A ForSyDe process constructor</td>
<td>49</td>
</tr>
<tr>
<td>5.4</td>
<td>A ForSyDe composite process wrapping two leaf processes</td>
<td>50</td>
</tr>
<tr>
<td>5.5</td>
<td>A ForSyDe XML representation</td>
<td>51</td>
</tr>
<tr>
<td>6.1</td>
<td>Stalling multiplication on a Nios II/f core</td>
<td>60</td>
</tr>
<tr>
<td>6.2</td>
<td>Non-stalling multiplication on a Nios II/f core</td>
<td>60</td>
</tr>
<tr>
<td>6.3</td>
<td>The individual grant masks and combinational logic of the TDM arbitrator</td>
<td>68</td>
</tr>
<tr>
<td>6.4</td>
<td>The grant shift process in the TDM arbitrator</td>
<td>70</td>
</tr>
<tr>
<td>6.5</td>
<td>The slot length counter in the TDM arbitrator</td>
<td>70</td>
</tr>
<tr>
<td>6.6</td>
<td>Example of the text-based input to the system creation flow</td>
<td>73</td>
</tr>
<tr>
<td>6.7</td>
<td>Instantiation of a Nios II/e processor core</td>
<td>74</td>
</tr>
<tr>
<td>6.8</td>
<td>Instantiation and connection of an on-chip memory element</td>
<td>75</td>
</tr>
<tr>
<td>6.9</td>
<td>Instantiation of an on-chip memory element and its TDM arbitration details</td>
<td>76</td>
</tr>
<tr>
<td>6.10</td>
<td>Instantiation and connection of an off-chip SRAM element</td>
<td>76</td>
</tr>
<tr>
<td>6.11</td>
<td>Instantiation and connection of a UART element</td>
<td>77</td>
</tr>
<tr>
<td>6.12</td>
<td>Instantiation and connection of a performance counter element</td>
<td>77</td>
</tr>
<tr>
<td>6.13</td>
<td>Instantiation and connection of a PIO module</td>
<td>78</td>
</tr>
<tr>
<td>6.14</td>
<td>Instantiation and connection of a DMA controller core</td>
<td>78</td>
</tr>
<tr>
<td>6.15</td>
<td>Instantiation and connection of a FIFO memory element</td>
<td>79</td>
</tr>
<tr>
<td>6.16</td>
<td>The chip select signal for an arbitrary number of master ports</td>
<td>85</td>
</tr>
<tr>
<td>6.17</td>
<td>An example of the XML-based output format of the DSE phase</td>
<td>90</td>
</tr>
<tr>
<td>6.18</td>
<td>The data structure used to maintain FIFO data</td>
<td>93</td>
</tr>
<tr>
<td>6.19</td>
<td>The CPU-specific header information</td>
<td>93</td>
</tr>
<tr>
<td>7.1</td>
<td>Annotating SWEET input data</td>
<td>99</td>
</tr>
<tr>
<td>A.1</td>
<td>DSE phase output describing the simple process network</td>
<td>129</td>
</tr>
</tbody>
</table>
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALF</td>
<td>Artist Flow Analysis</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetical-Logical Unit</td>
</tr>
<tr>
<td>AMC</td>
<td>Analyzable Memory Controller</td>
</tr>
<tr>
<td>BCEC</td>
<td>Best-Case Execution Cycles</td>
</tr>
<tr>
<td>BCET</td>
<td>Best-Case Execution Time</td>
</tr>
<tr>
<td>BSP</td>
<td>Board Support Package</td>
</tr>
<tr>
<td>CCSP</td>
<td>Credit-Controlled Static-Priority</td>
</tr>
<tr>
<td>CompSOC</td>
<td>Composable and Predictable Multi-Processor System-on-Chip</td>
</tr>
<tr>
<td>DCT</td>
<td>Discrete Cosine Transformation</td>
</tr>
<tr>
<td>DI</td>
<td>Domain Interface</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DSE</td>
<td>Design Space Exploration</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain-specific Language</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In, First Out</td>
</tr>
<tr>
<td>Flit</td>
<td>Flow Control Unit</td>
</tr>
<tr>
<td>ForSyDe</td>
<td>Formal System Design</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
</tr>
<tr>
<td>FPP</td>
<td>Fixed Priority Preemptive</td>
</tr>
<tr>
<td>HAL</td>
<td>Hardware Abstraction Layer</td>
</tr>
<tr>
<td>HRT</td>
<td>Hard Real-Time</td>
</tr>
<tr>
<td>HSDF</td>
<td>Homogeneous Synchronous Data Flow</td>
</tr>
<tr>
<td>ICRTA</td>
<td>Intra-Core Real-Time Arbiter</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>JPEG</td>
<td>Joint Photographic Experts Group</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>JTAG</td>
<td>Joint Test Action Group</td>
</tr>
<tr>
<td>MCU</td>
<td>Minimum Coded Unit</td>
</tr>
<tr>
<td>Merasa</td>
<td>Multi-Core Execution of Hard Real-Time Applications Supporting Analyzability</td>
</tr>
<tr>
<td>MoC</td>
<td>Model of Computation</td>
</tr>
<tr>
<td>MPSoC</td>
<td>Multiprocessor System-on-Chip</td>
</tr>
<tr>
<td>NHRT</td>
<td>Non-Hard Real-Time</td>
</tr>
<tr>
<td>NI</td>
<td>Network Interface</td>
</tr>
<tr>
<td>NoC</td>
<td>Network-on-Chip</td>
</tr>
<tr>
<td>PIO</td>
<td>Programmed Input/Output</td>
</tr>
<tr>
<td>Predator</td>
<td>Predictable SDRAM Memory Controller</td>
</tr>
<tr>
<td>SDF</td>
<td>Synchronous Data Flow</td>
</tr>
<tr>
<td>SDRAM</td>
<td>Synchronous Dynamic Random Access Memory</td>
</tr>
<tr>
<td>SMT</td>
<td>Simultaneous Multi-Threading</td>
</tr>
<tr>
<td>SOPC</td>
<td>System-on-a-Programmable-Chip</td>
</tr>
<tr>
<td>SPL</td>
<td>Split Phase Load</td>
</tr>
<tr>
<td>SRAM</td>
<td>Static Random Access Memory</td>
</tr>
<tr>
<td>SUSAN</td>
<td>Smallest Unvalue Segment Assimilating Nucleus</td>
</tr>
<tr>
<td>SWEET</td>
<td>Swedish Execution Time Tool</td>
</tr>
<tr>
<td>Tcl</td>
<td>Tool Command Language</td>
</tr>
<tr>
<td>TDM</td>
<td>Time Division Multiplexing</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver/Transmitter</td>
</tr>
<tr>
<td>UBD</td>
<td>Upper Bound Delay</td>
</tr>
<tr>
<td>USAN</td>
<td>Unvalue Segment Assimilating Nucleus</td>
</tr>
<tr>
<td>VHDL</td>
<td>Very-High-Speed Integrated Circuits Hardware Description Language</td>
</tr>
<tr>
<td>WCCC</td>
<td>Worst-Case Communication Cycles</td>
</tr>
<tr>
<td>WCCT</td>
<td>Worst-Case Communication Time</td>
</tr>
<tr>
<td>WCEC</td>
<td>Worst-Case Execution Cycles</td>
</tr>
<tr>
<td>WCET</td>
<td>Worst-Case Execution Time</td>
</tr>
<tr>
<td>XCBA</td>
<td>Inter-Core Bus-Arbiter</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
1

Introduction & Background

This first chapter of this thesis will introduce the area of embedded computing and real-time systems as well as the unique challenges their design and programming presents. Further, it will highlight the specific problem this thesis aims to solve, the methods applied in the process and give an outline of this document itself.

1.1 Embedded systems

Over the past decades, computer systems have been introduced into nearly every aspect of modern society. From these computers, only the smallest part is formed by personal tools like desktop computers, notebooks and mobile phones. The majority of computers in use today is embedded into other devices, in which they perform key functions of data collection, control and communication. These devices can be home appliances, medical devices like artificial pacemakers and patient monitoring instruments, automotive components like anti-lock brakes and engine controls or satellites. This class of computers, which forms 98% of microprocessors produced worldwide [Ebert and Jones 2009], called embedded systems, spans the widest range of applications and performance. Unlike multi-purpose computers, they are designed and highly optimized for one single or a small number of tasks [D. A. Patterson and J. L. Hennessy 2008, p. 6]. Additionally, because of their utilization in mass-produced devices, their design focus lies on low cost and power efficiency.

1.1.1 Real-time systems

Real-time systems are a category of embedded systems that must react to signals from their environment within precise time constraints, which makes them a special case of reactive systems [Berry 1989]. They therefore share the characteristic that their functionality and interaction with the environment does not only have functional but also reaction requirements [Henzinger 2008], meaning that their correct operation does not only depend on the results of their operation but also on the timing of these outputs.
1.1. Embedded systems

An example for a real-time system is a machine at the conveyor belt of a bottling plant, controlled by an embedded system. Its function is to cap each bottle that passes on the belt while keeping track of the speed of the belt and the position of the bottles. Both factors have to be considered before the embedded systems can cap a specific bottle: what is its exact position and how fast is it moving? If the machine is too fast or too slow, one or several bottles remain open and the whole conveyor belt has to be stopped. To avoid this, the embedded system controlling the machine has to be designed to handle speed changes and absent bottles correctly and with correct timing. Therefore, the individual tasks and functions inside the embedded system are created with an assigned deadline in mind, i.e. a point in time at which their results must be present to the environment or to another task waiting for it. For example, a bottle must be correctly detected in such a way that the capping mechanism has enough time to lower a cap and tighten it.

1.1.2 Hard real-time systems

An important subset of real-time embedded systems controls safety-critical devices and components. These can perform, in case of an artificial pacemaker, tasks like the monitoring of patient life signs and, if necessary, admitting electrical impulses to the heart of the wearer. Further examples include monitoring and control functions in cars, planes and space crafts, as well as the automatic control of railroad switches and power plants. An embedded system has Hard Real-Time (HRT) characteristics if a failure to complete its tasks before their assigned deadlines leads to a fatal fault of the system [Liu 2000].

1.1.3 Soft real-time systems

While a failure to produce results in a timely fashion leads to fatal consequences in a hard real-time system and must therefore be avoided at all costs, a soft real-time system is characterized by the fact that a timing violation of a task is merely undesirable. An example for a soft real-time system can be found by considering most video processing systems. These systems are designed to be fast enough to process video frames at a chosen rate. However, if the device occasionally misses one frame due to heavy load, the viewer will not or only barely recognize it.

Another example is a cruise-control system in a car. The purpose of these systems is to keep the car at a certain speed set by the driver. To accomplish this, the system periodically samples the current speed of the car and accelerates or decelerates the vehicle accordingly to reach the target speed. If the cruise-control software sporadically fails to sample the speed of the car, the system can use the previously measured value, given that the original sampling frequency is high enough. If it, however, fails to measure too large a number of consecutive samples, the system as a whole fails to complete its function and the consequences for the vehicle and its occupants could be severe.
As can be seen from this example, the distinction between hard and soft real-time systems can become unclear for the area of applications where a small number of misses does not lead to a total failure, but missing more than a few may lead to complete and catastrophic system failure. These embedded systems therefore fall under a different category called firm real-time systems [Laplante 2004].

1.1.4 Designing embedded systems

Due to their characteristics and areas of utilization described above, the design of embedded systems has multiple, partially conflicting goals. Because they are most commonly used in mass-produced devices, one design focus is on low design, manufacturing and maintenance costs while ensuring correct functionality and timing, power efficiency and a short time-to-market.

During the design process of embedded systems, the starting point is an initial specification, comprised of the intended functionality of the system and the set of constraints on both the design process and the final product, e.g. the power consumption of the product and its manufacturing costs. This specification is written at a very high abstraction level with a large number of possible implementations while the final step in the design process in one specific implementation of the system at the lowest possible level of abstraction. To overcome this abstraction gap, the design process consists of a large number of decisions that lower the abstraction level and limit the possible design space regarding the choice of hardware and software. Figure 1.1 illustrates this top-down design process.

![Figure 1.1: The top-down design process, taken from [Sander 2003].](image)

While several of these refinement steps during the design process are today carried out
1.1. Embedded systems

automatically by tools that ensure correctness, e.g. software compilation and hardware synthesis, the majority have to be performed by the system designers. However, each of these manual steps introduces a new element of uncertainty and a possible source of error into the process and therefore into the final product. Today, more than 80% of development resources of embedded systems are spent on testing and analysis due to the growing complexity of both the hardware as well as the software components [E. Lee 2008]. On the hardware side, it can be observed that the cost of utilizing multiple cores and processors on a single chip has decreased to a level that their integration into a wide range of embedded systems has become the most affordable option for system designers [Levy and Conte 2009]. The same is valid for the software development side of embedded systems design: new applications have to be designed with parallelism in mind to be able to fully utilize the possibilities of modern hardware architectures, while legacy code has to be adapted for the same purpose.

This poses a major challenge for the design of real-time systems, whose correct functionality and timing have to be ensured before releasing the product. To mitigate this, two fundamentally different approaches have been developed in the past decades: a measurement-based and a formal, or static, approach. The measurement-based approach uses an implemented system prototype or a simulation model of it, as well as a set of input data to try to identify errors in the system and generate an approximation of its timing behavior. Although it is the most widely-used method to test embedded systems [Lokuciejewski and Marwedel 2011], it is inherently error-prone and slow, because only a subset of input data can be used to measure the behavior of the system and every test run requires a simulation of the hardware and software or the system itself. This method is therefore not able to produce safe conclusion about correctness and the timing behavior of the tested components.

The static approach to the testing of embedded systems forms a fundamentally different method. Unlike the measurement-based approach, it does not rely on the variation of input data and simulation to gather information about errors and timing, but is based on a mathematical model created from both the hardware as well as the software of a system. Using this method, an automated tool flow can be established, that analyzes the source code produced by the system designer and identifies both errors in the program flow as well as exact timing information about the developed system. A basic requirement of this testing method is the existence of a mathematical model of both the hardware and the software. However, the difficulty of creating the required models of modern hardware and software systems has increased tremendously with the rise of multiprocessor systems and architectural features such as caches, pipelined instruction execution and hardware multithreading. These features, while increasing performance and power efficiency, pose unique challenges to the analysis and testing of embedded systems. Both methods will be further explained in Section 2.2.
Chapter 1. Introduction & Background

During recent years, research has been performed that aims to solve this standing problem to conduct efficient and correct error testing and analysis of embedded systems. These research endeavors developed new design methodologies, that share the trait of being able to create embedded systems which are correct-by-design. As mentioned before, during the design of embedded systems, a number of refinement steps bridging levels of abstraction can be performed automatically by software tools. The most common of these steps are the automatic compilation of software from high-level languages to machine code and the automatic synthesis of hardware from a hardware description language like VHDL or Verilog to the gate level. Both these processes are comprised of a large number of subprocesses that have been the subject of research and analysis for decades and have been shown to be almost free of errors by extensive testing and the application of formal analysis to prove their functionality. The aim of the design methodologies currently in development is to widen the gap in abstraction that can be bridged by software and automated flows. Examples are the automatic synthesis of hardware description code from a high-level programming language by high-level synthesis tools or the extraction of hardware component requirements from a high-level description and their automatic allocation from a set of standard components to suit the needs of the described application.

One such methodology is Formal System Design (ForSyDe), developed at KTH by Sander et al. [Sander 2003]. Its functionality, components and the basics of designing a system using ForSyDe will be explained in Chapter 5.

1.2 Problem

In order to create a correct-by-construction design flow for real-time embedded systems, a Multiprocessor System-on-Chip target platform needs to provide a number of features which set it apart from traditional hardware architectures. Due to the varying demands of applications to be implemented on the platform, it needs to be conceived as a template offering the possibility to create a wide range of architecture instances using a set of components provided by the platform, such as processors, memories and input and output facilities. Furthermore, to be able to move timing concerns to the architecture level, it needs to provide the possibility to extract exact timing information of a complete system, including computation on individual processors as well as communication between multiple processors and shared resources.

A part of the current research efforts around the ForSyDe design methodology aims to create such a correct-by-construction design flow for real-time embedded systems. Using ForSyDe a system is described on a high abstraction level while the resulting implementation is generated either by the automatic stepwise application of design transformations until a low level of abstraction is reached or by mapping the high-level
system description to a Multiprocessor System-on-Chip (MPSoC). In the case of the latter, the lowest level of abstraction describes the multiprocessor hardware to be implemented as well as the compiled application code with all required functionality. In combination with a platform template described above, ensuring timing predictability of applications to be executed on it while offering the possibility to be adapted in a number of aspects such as the amount of processing elements and the memory layout, it is possible to create a complete correct-by-construction design flow for real-time embedded systems.

This thesis aims to create such a Multiprocessor System-on-Chip template based on the Altera Avalon Switch Fabric [Altera 2006a] and Nios II soft processors [Altera 2011a], modified to support the static prediction of timing behavior and take the first steps on the way to integrating it with the current status of the development of the ForSyDe system design flow.

1.3 Method

To be able to understand the underlying problem of timing predictability and extend the ForSyDe functionality, a literature study into different forms of timing predictability and existing hardware platforms with this functionality has been conducted. Furthermore, the possibility to modify Altera components has been analyzed: the processing elements, the interconnection network as well as the memory connection. Using these components, a tool flow was developed and integrated into the existing ForSyDe flow to automatically create the hardware structure for a given set of ForSyDe processes. Finally, sample applications were created and shown to both produce correct results and exhibit the predicted timing behavior.

1.4 Outline

The following second chapter of this thesis will introduce the concept of timing predictability in hardware and software and the challenges connected with its realization. Using this information, the third and fourth chapters will present a number of current approaches to the creation of time-predictable hardware and software architectures and how they address the issues introduced in the second chapter.

The fifth chapter will offer an insight into ForSyDe, an approach that offers solutions to a range of problems in the area of the design of embedded systems which also forms the surrounding framework for this thesis. This chapter will present the concept of models of computation, the idea of hardware/software codesign and the correct-by-design idea that stands behind ForSyDe.

The sixth chapter will present the introduction of time predictability into Altera Nios II-based designs. By analyzing the individual parts of these systems for the degree of timing
variation they exhibit, the necessary modifications made to support the determination of
timing bounds as well as their integration into the ForSyDe design flow.

The seventh chapter will present the tests performed to verify the functionality
and the predictability of the created hardware architecture as well as two example
ForSyDe applications that have been developed for the platform and tested on a Field-
Programmable Gate Array (FPGA).

The eight chapter will conclude this thesis by giving a summary and final remarks to
possible extensions to the developed architecture as well as the ForSyDe design flow.
1.4. Outline
2
Predicting Timing Behavior

The following chapter will introduce the concept of timing analysis for embedded systems, its importance in the field of hard real-time systems as well as the problems that emerged with the increase in system complexity and the rise of multi-core computing.

2.1 The need for timing analysis

The design of standard computer architectures, both in hardware as well as software is based on the paradigm to make the common fast and the uncommon case correct [J. Hennessy and D. Patterson 2011]. While this approach has been successful in the creation of systems with high data throughput, it also created the requirement to implement increasingly complex mechanisms to facilitate increases in performance. However, the more complex these architectural features are becoming, the harder it is to predict their timing behavior for all possible situations and use cases. Section 2.2.1 gives an overview of the most common enhancements made to optimize the performance of standard computer architectures and how they affect the analyzability of the system as a whole.

Due to this, the design of hard real-time systems has unique requirements in its choice of hardware and software components as the possibility to analyze the system timing is a basic requirement to guarantee correct functionality in all possible load situations. These include the common use cases of a system as well as all possible combinations of input signals that influence the system behavior. It has to be guaranteed that the timing of the system follows the specifications after which it was developed in every case: its timing behavior therefore has to be able to be predicted at design time. The timing prediction of a system that is needed to guarantee correct functionality in every case, is the prediction of the worst-case timing behavior, i.e., the execution delay of the tasks during the highest possible load. For hard real-time systems, this delay must not exceed a certain value and its prediction therefore marks a basic requirement of the design of such systems. Similarly, the Worst-Case Execution Time (WCET) of a single task is defined as the longest execution time of a task that could ever be observed when running on the system hardware [Ermedahl 2003]. The same is valid when including the
interconnection structures between multiple processing elements and shared resources into the consideration: transfers from one processor might behave differently due to conflicting operations from other processors. It is the goal of timing analysis to provide estimates for the worst-case execution times of a task running on a given system in order to evaluate the schedulability of task sets [Seto et al. 1996; Stigge 2007]. For the same reason, other timing measures can also be useful and be part of the analysis, e.g., the Best-Case Execution Time (BCET) of a task, which is defined as the shortest execution time of a task that could be observed on a system. As mentioned above, the more complex a software and hardware system becomes, the harder it is to model and analyze their behavior for all possible situations. For the process of timing analysis, this leads to difficulties in the prediction of exact timing bounds. Consequently, the most important quality characteristic of these execution time measures, or bounds, is their proximity to their real counterparts.

Worst-case and best-case performance. The worst-case and best-case execution times define the maximal and minimal time interval between the timing events under all admissible system and environment states [Thiele and Wilhelm 2004]. All possible execution times that occur due to variations of input data and the environment of the system lie between these two values.

Upper and lower bounds. The upper and lower bounds are factors that bound the worst-case and best-case timing behavior. The extraction of these values is the goal of timing analysis.

2.2 Calculation of execution time bounds

Using the definitions presented in the previous section, the following will shed light on the two main approaches to determine execution time bounds.

2.2.1 Problems and requirements

As mentioned in the previous section, to make timing analysis of a system possible, its behavior as a whole and of all its individual components has to be able to be modeled. This process involves the abstraction from the real hardware and software towards a simplified mathematical model that can be the basis for a number of analysis techniques. This abstraction step is possible for simple hardware components but gets increasingly harder to take, the more complex the components become [Schoeberl 2009]. In addition to that, in a multiprocessor system, the execution time of an individual task does not only rely on the task itself but on all tasks running on all processors. The reasons for this is, that even if a single processor and memory can be modeled to analyze a task running on
it, additional processors can cause access to the same memory, creating resource conflicts and thereby invalidating the model.

Research into the timing analysis of digital circuits is a well established field and produced widely accepted methods to analyze the timing behavior of electrical signals in integrated circuits, which are an integral part of the design process of hardware systems [Kukimoto, Berkelaar, and Sakallah 2002]. Due to the nature of their operation as clocked systems being built from simple components, timing analysis for integrated circuits can be performed statically, i.e., neither depending on specific values of signals to the system nor specific usage patterns. As a result of this timing analysis, the timing behavior of all electrical signals in the system is analyzed and checked for violations towards the specification of the transistors of the target platform as well as towards clock timing. However, unlike during the evaluation of the electrical wire delay, the behavior of hardware systems that implement complex algorithmic behavior cannot be broken down to a simple model. The behavior of these systems is influenced by a large number of parameters during the design of the system as well as during its operation. The following presents a number of the most commonly implemented features of modern hardware architectures that have an adverse effect on timing analyzability.

**Pipelined instruction execution.** Pipelined instruction execution is the most common feature to increase the clock frequency of processors by reducing the critical path. However, fitting instructions into each other creates dependencies between them which can cause structural as well as data hazards that need to get resolved to enable correct functionality. A common method to resolve a hazard is to stall the pipeline, i.e., stopping newly loaded instructions and letting older instructions finish their operation before resuming. This effectively increases the execution time for instruction should these hazards occur and introduces dependencies between instructions that complicate the analysis of a program stream [Engblom and Jonsson 2002].

**Caches.** Similar to pipelines, caches introduce dependencies between instructions that can cause inconsistencies in the execution time of instructions. The reason for this is that the loading time of data, regardless of the choice of cache organization and replacement scheme, depends on the location of the data in the main memory and current state of the cache, which in turn is determined by all previous instructions that loaded data from the main memory. Additionally, cached data can be the cause of timing anomalies [Lundqvist and Stenstrom 1999; Reineke et al. 2006; Wilhelm et al. 2008]. A timing anomaly in the context of cache memories occurs, when a cache operation causes an increase in the execution time of the current instruction decreases the overall execution time of the sequence of instructions. The opposite effect, a decrease in local but increase in overall execution time, is also considered a timing anomaly.
2.2. Calculation of execution time bounds

Figure 2.1a illustrates this problem. The latter case, however, can be ignored in the context of timing analysis as they only accelerate a task, therefore making a WCET prediction less accurate but not invalid. Additionally, caches intensify the difficulties in predicting timing on multiprocessor systems with shared memories and shared caches due to the increased timing cost of conflicts (so called storage interferences) between processors and the increase in complexity of a model describing the system behavior to a degree at which it becomes infeasible as a basis for timing analysis.

(a) A timing anomaly possible to occur when multiple processes are using two shared resources. A-E are processes accessing these resources and their dependencies are displayed using arrows. The upper case exhibits a worse local execution time for A but a shorter global execution time. The reason is that process B can be scheduled earlier in the second case but blocks its resource, which in turn blocks the whole schedule due to the dependencies between the processes.

(b) A possible timing anomaly when using prefetching caches in combination with speculative instruction execution. A mispredicted branch results in unnecessary instruction fetching which destroys the cache state. If, however, the first instruction to be fetched results in a cache miss, the correct branch condition will be evaluated before the cache state is destroyed.

Figure 2.1: The two types of timing anomalies when utilizing caches and speculative instruction execution. Taken from [Reineke et al. 2006].
Branch prediction. Branch prediction enables the filling of the pipeline across the boundaries of a branch in the sequence of instruction by trying to predict the branch outcome and speculatively loading instructions at the predicted branch target. This inherently causes problems for timing analysis as the real branch target depends on the actual data processed by the instruction sequence and a branch prediction mechanism can only guess its outcome, causing stalls of the pipeline and therefore possible violations of the predicted timing at possibly every branch [Chang et al. 1994]. Additionally, branch prediction hardware and speculative execution of instruction can lead to the occurrence of timing anomalies, further complicating the modeling of the system behavior. Figure 2.1b illustrates the possibility of timing anomalies when using branch predictors.

Instruction-level parallelism, chip multi-threading and out-of-order execution. Modern processor architectures dynamically reschedule instructions to increase the degree of instruction-level parallelism and, if possible, execute multiple instructions simultaneously. For example, a processor like the Pentium 4 is able to simultaneously track up to 100 instructions [D. A. Patterson and J. L. Hennessy 2008] and reschedules them as needed to maximize performance. As Schoeberl [2009] states, the creation of an abstract model describing this behavior is almost impossible, therefore making these processor features infeasible for usage in hard real-time systems.

2.2.2 Existing approaches to predicting timing behavior

The following section briefly describes the two main approaches to predicting the timing of a task or a set of tasks on a given hardware system and how these are influenced by the aforementioned complex hardware mechanism to improve system performance.

The first and most commonly used technique to determine task timing is based on measurements of the task performance either on the target hardware or on a simulation model. This approach draws conclusions from repeated runs using sets of input data to try to determine the worst-case and best-case execution times [Lim et al. 1995]. The second approach, static program analysis is a generic method to determine the behavior of a task without executing it. This method relies on mathematical models describing the possible program flows on an abstract model of the underlying hardware. Unlike measurement-based approaches, these methods are independent of input data and can therefore cover all possible cases of the program flow.

However, both approaches introduce an amount of uncertainty to the timing predictions they are able to give. Measurement-based methods, regardless whether they utilize a simulation model or the real hardware, are not able to cover all possible combinations of input values and their behavior over time. Consequently, unless all possible sets of input data and therefore execution paths through the program flow and system context are measured, these methods can only give a coarse estimate of
2.2. Calculation of execution time bounds

Distribution of times

Figure 2.2: Histogram of timing analysis result for a fictional program. The abscissa represents the execution time of the program while the ordinate shows the amount of test runs during which this particular execution time was measured. Adapted from [Ermedahl 2003].

the real timing behavior. To mitigate this problem, it is a common step to measure the execution times of a program for a sensible set of input data and add a safety margin to the determined execution time to avoid underestimation [Lokuciejewski and Marwedel 2011]. Due to this, measurement-based methods are not able to determine guaranteed lower and upper bounds for the task execution time. Figure 2.2 illustrates the problems encountered when determining timing information using measurements. As can be seen there, all possible execution times of a task lie between the best-case and the worst-case execution time and only a small amount of possible execution flows lead to execution times close to these values. The gray bars show the amount of program flows that lead to the execution time at that point while the white bars show the amount of measured program execution that lead to this execution time. In the context of this example, out of all measurements, none reached the best-case or the worst-case execution time, thereby leading to uncertainties when drawing conclusion from these test results. Further, all estimates between the real best-case and worst-case execution times have to considered as unsafe.

Static methods, on the other hand, do not suffer from this problem. By analyzing its control flow and using safe abstractions of the set of possible input values, static methods are able to determine the Worst-Case Execution Path (WCEP) throughout a program. From this, it is possible to determine guaranteed and safe worst-case execution times. This step, however, as mentioned above, heavily relies on a model of the underlying hardware platform to generate timing bounds. If the model is correct and precisely specifies the
hardware, the generated timing bounds are safe and tight, i.e., they neither overestimate nor underestimate the real values. If a model, however, is in itself based on conservative estimations about the timing behavior of the underlying hardware because a precise model cannot be created or is too complex to be a basis for calculations, the less precise the timing bounds of static analysis will become. The difference between the real and determined WCET is often referred to as the pessimism of the analysis estimation.

In summary, it can be said that the two main criteria for evaluating the two presented approaches to timing analysis are their safety: does the method calculate a guaranteed bound or does it rely on estimations, and their precision: how close to the actual best-case and worst-case execution times are the determined bounds. In hard real-time systems, a feature that increases average performance but that cannot be modeled to decrease WCET bounds, can be considered useless.

Because of the features of modern hardware and the difficulties described in Section 2.2.1 and in creating precise abstraction models, static analysis methods are most commonly used in academia and are not widely adopted in industry. Nevertheless, the possibility to create safe and tight bounds when using hardware customized for timing analysis lead to an adoption in highly safety-critical domains such as the development of systems for automotive [Byhlin et al. 2005] and avionic [Souyris et al. 2005] applications.

The following chapter will present current approaches to enable the creation of high-performance hardware architectures that retain the possibility to create abstraction models feasible for static program analysis.
2.2. Calculation of execution time bounds
3 Design of Time-Predictable Hardware Architectures

As the previous chapter discussed, modern hardware architectures are designed with respect to their average-case performance. The following will present a number of approaches to mitigate the impact of these design choices on the calculation of worst-case execution times.

3.1 Requirements for predictability

The timing behavior of individual components of a hardware architecture has varying degrees of influence on the predictability of the complete system. In the following section, the main components and their requirements to maximize timing predictability are presented.

3.1.1 Principles for a predictable processor

In order to design a predictable processor, Berg, Engblom, and Wilhelm [2004] define a set of basic requirements that have to be fulfilled to achieve the highest degree of analyzability and predictability.

**Minimal variation.** In order to make the analysis of a processor more precise, variations in instruction timing should be avoided. Both best-case and worst-case performance are important metrics, so optimizations that accelerate the average case but show large variation in performance should be eliminated as they lower the precision of an architectural model. For example, the implementation of a multiplication can be accelerated in the case of certain operands, e.g. by shifting the operand in case of a multiplication with a power of two. To precisely model this behavior, an additional dependency will have to be introduced that describes the relation of the instruction timing and the operands, effectively increasing the complexity of the analysis.

**Non-interference.** To avoid adding complexity to the model of a processor, individual components have to be decoupled in a way that the behavior of one component should
3.1. Requirements for predictability

not influence the behavior of other components. If this is not the case, components cannot be analyzed individually and a model has to accommodate these influences, increasing its complexity with each new component. This contradicts the principle presented in Section 2.2.2 which describes that an analysis uses abstractions to model the system behavior as close as possible. Such an abstract model cannot be created when a large number of interferences between individual processor components are present.

An example for this behavior is the functionality of a branch prediction mechanism, which changes the state of data and instruction caches according to its current prediction. If this is the case, a separate analysis of cache and branch prediction mechanism is not possible and the complexity of an analysis model increases significantly.

Determinism. To enable analyzability, processors should be deterministic in the following sense:

- **Concrete determinism** In every case, the processor hardware has to behave deterministically. The design needs to be fully synchronous to enable static timing analysis of the electrical circuits and all operations, e.g. on a bus or inside an ALU, have to be performed in designated numbers of cycles.

- **Observable behavior** The timing of an operation should only depend on factors observable to the programmer and the analysis. If this is not the case, the creation of an abstraction model becomes impossible. For example, if a static analysis tool keeps track of possible cache replacement operations, a round-robin block replacement counter that increments only on cache hits, cannot be modeled or a worst-case timing has to be assumed for every cache operation.

- **Abstract determinism** This type of determinism is a consequence of minimal variation and non-interference described above. While modeling the behavior of a hardware system, there is a need to make abstractions to keep the complexity of the model as low as possible. Abstract determinism describes that these abstractions have to have a minimal effect on the precision of timing analysis, i.e. all abstraction have to have a bounded effect to be able to tightly model the behavior of the hardware.

Comprehensive documentation. To be able to fully understand and precisely model the behavior of a processor, it is necessary to have access to the complete documentation. In the best case, this documentation is cycle-accurate and describes every instruction and their pipeline profiles as well as all exceptions and corner cases.

3.1.2 Principles for a predictable memory controller

While the principles stated above are also valid for memory controllers, there are a number of special requirements that have to be fulfilled to create a memory controller
that supports tight timing analyzability.

The main goals when designing a predictable memory controller are to provide a guaranteed bandwidth and a bounded maximum latency for individual requests. While this does not pose a challenge in the design of controllers for Static Random Access Memory (SRAM), which has a fixed access latency for every request, it has been the subject of multiple research efforts to design an Synchronous Dynamic Random Access Memory (SDRAM) controller that offers these features [Akesson, K. Goossens, and Ringhofer 2007; Bhat and Mueller 2010; Paolieri, Quinones, et al. 2009].

Due to the mode of operation of SDRAM, the timing of individual accesses always depends on surrounding operations [D. A. Patterson and J. L. Hennessy 2008, p. 473]. For example, every first access to a memory bank requires an activate command to load the requested row into the row buffer. Subsequent accesses to the same bank are performed without this preceding command and are therefore faster. Accessing a different bank requires the memory controller to first precharge the previous bank and then activate the requested bank to serve the request. Additionally, since SDRAM contents are non-permanent, the memory controller needs to periodically issue a refresh command to the underlying hardware to retain the stored data.

To enable the determination of tight timing bounds for memory accesses, a predictable SDRAM controller therefore has to guarantee a fixed latency for every request and, if multiple readers and writers are connected, arbitrate their requests in order to ensure a minimum bandwidth. Following these requirements, Chapter 4 presents an implementation of a highly predictable memory controller.

3.1.3 Principles for a predictable interconnection structure

One of the main issues for predictability in multiprocessor systems is the communication infrastructure shared between processors, memories and peripheral devices. Traditionally, the interconnection used for systems with a low number of processors is bus-based, i.e. a single communication link that is shared by all devices but can only be used by a single sender at a time. With the increasing number of processor cores on a single chip, buses form a significant bottleneck and are largely replaced by on-chip networks called Networks-on-Chip (NoCs) [Hemani et al. 2000]. This medium allows simultaneous communication between any device on the chip with any other chip resource, thereby effectively eliminating the data transfer bottleneck imposed by bus-based solutions. However, both methods share the inherent challenge in the design of predictable systems, that the timing of data transfers of a single core does not solely depend on its own operation but on the transfers executed by all cores on the chip. A timing analysis would therefore have to simultaneously take all cores into account when predicting the timing of communication operations between cores and chip resources. This would increase the complexity of an analysis model to an unfeasible degree. To mitigate this problem
3.2 Definitions of the term predictability

Due to the challenges presented in the previous sections, a number of attempts have been made to define the term predictability in order to identify novel ways to approach the creation of predictable embedded systems.

3.2.1 Predictability through determinism

In Henzinger [2008], the author defines the behavior of a real-time system not only by the values the system computes from its input signals, but also by the times these values are output by the system, thereby promoting the timing behavior to an equally important characteristic as the system functionality itself. Based on this, he extends the notion of determinism to a combination of the functional and the temporal properties of a system and uses this definition to present the following two approaches to the creation of predictable system.

Predictable systems built entirely from deterministic and predictable parts. The first approach to the creation of fully predictable systems is to build them entirely from predictable and deterministic components that adhere to the definition of determinism stated above. Their functional and temporal behavior must be fully understood and predictable. For the example of a processor, the timing of each instruction has to be fixed and known at design time, including all accesses to memories and communication...
Chapter 3. Design of Time-Predictable Hardware Architectures

channels. However, for two reasons, it is his opinion that this fully deterministic approach to the design of predictable systems cannot provide a generally acceptable solution: first, every system will suffer from residual sources of non-determinism that will be impossible to remove by the system designer, e.g., the failure of hardware components or fluctuations in the electrical behavior of the hardware implementation. Second, by reducing the choice of components during the construction of the system to only fully deterministic ones, the design process becomes too restrictive and ignores non-deterministic components, whose unpredictable behavior can be hidden underneath abstraction layers that provide predictability. This leads to the approach explained in the next section.

Predictable systems using deterministic abstraction layers. As explained in Section 2.2.1, a large number of features of modern processor architectures severely decrease the predictability of these hardware components by introducing variations as well as anomalies into their timing behavior. However, Henzinger states that it is not feasible for the design of high-performance embedded systems to completely avoid these components but that their non-deterministic timing behavior should rather be hidden underneath layers that provide deterministic abstractions from the unpredictable features of the underlying hardware. These layers can be based on hardware, on software or on a combination of both. An example for a hardware-based layer is presented in Di Carlo et al. [2010], a system using partial reconfiguration of FPGA to mask electrical failures in embedded systems that can be caused by radiation or other environmental influences. An example for an abstraction layer based on software are the family of synchronous programming languages [Halbwachs 1992], that hide the implementation details of process timing and communication by providing support for concurrent programming built into the language.

Based on this approach, it is possible to identify four sources of non-determinism in a system, of which only one has to be avoided in the design of predictable systems:

i) input non-determinism
ii) unobservable implementation non-determinism
iii) don’t care non-determinism
iv) observable implementation non-determinism

While all four sources of non-determinism can be found in embedded systems, Henzinger argues that only the fourth kind, observable implementation non-determinism is harmful to the predictability of a system and therefore has to be avoided.

The first source of non-determinism, input non-determinism, describes that the environment of a system is free to behave in any way. As reactive systems are built

---

1 As Chapter 6 shows, this is the approach taken by this thesis
to react to their environment, this non-determinism is an integral part of the view on such systems. The notion of determinism of a system itself is therefore disconnected from the determinism of its environment but encompasses the idea that a system is considered time-deterministic if for each timed stream of input data from the environment, the output stream is the same.

The second source, unobservable implementation non-determinism, describes the omission of non-deterministic implementation details by a layer providing determinism, as explained above. As a consequence, the non-determinism and its results become unobservable to layers above, from which the system appears deterministic. This can include both functional as well as temporal parameters of a specific layer. An example is a system providing a sorting algorithm for a list of items. While the output of such a function must be a sorted list of the input items, the choice of sorting algorithm does not play a role for this concept of determinism, as it cannot be observed from the outside.

The third source of non-determinism, don’t care non-determinism, describes the existence of don’t care values in the output stream of a system. For example, a certain output value may be without significance if an input value asserts a certain condition. While this is a common concept in the design of computer systems, the definition of time determinism given above cannot easily accommodate it. If for an input value 0, the system may output either a 0 or a 1, the system behaves non-deterministically, as there are two possible outputs for the same input. However, the introduction of a separate don’t care value, denoted by ⊥, can solve this problem. For the same case, if the input of a system is 0, the output may either be 0 or 1, or simply ⊥. This transforms the event sequence from the non-deterministic \{ (0, 0), (0, 1) \} to \{ (0, ⊥) \}, which describes a deterministic sequence.

The fourth source of non-determinism describes the omission of implementation details that, in contrast to the second source, do influence the behavior of a system. If a layer omits details of the underlying system that have an influence of either the timing of the functionality of the generated results, it leads to observable implementation non-determinism, which is harmful in the context of system design. For example, if a task scheduler is free in the organization of tasks, then each stream of input data could generate a number of possible streams of output data, thereby rendering the system non-deterministic.

Of all four presented sources of non-determinism, the fourth kind is the only one that cannot easily be hidden underneath a deterministic layer, as this layer would need to consider all non-deterministic choices and their consequences to successfully hide them to the layers above. Therefore, during the design of highly predictable embedded systems, all observable implementation non-determinism has to be eliminated.

In the case of design of general computer systems, unobservable implementation non-determinism is a powerful tool to hide implementation details underneath layers
providing system services and functionality. For example, languages like Python provide the same functionality across different operating systems and hardware architectures, thereby hiding all implementation details from the software designer. Similarly, for the example of task scheduling, a possible solution to hide unobservable non-determinism of an underlying scheduling mechanism, is to promote the timing of tasks to a basic concept of the programming language used, as implemented in synchronous languages: To successfully hide non-deterministic scheduling and guarantee correct task timing, the results of operations can be held internally until they are needed, i.e., until the point in time is reached that is specified in the implementation of the operation.

In conclusion, Henzinger identifies the creation of deterministic abstractions on top of non-deterministic systems as the main challenge in the design of predictable embedded systems.

3.2.2 Is time-predictability quantifiable?

Schoeberl [2012] presents the argumentation that time predictability of a system in itself is a useful but incomplete property of a concrete system and that worst-case performance is the significant property. Predictability of a system alone does not enable a meaningful comparison to other predictable systems: the performance of the system as well as the pessimism of the underlying analysis method have to be considered to enable a relevant comparison.

For example, if a given task has a WCET of 1 000 cycles on a processor A and a WCET of 800 cycles on a processor B, it seems as if processor B is able to execute the task faster and shows better performance. However, this decision does not factor in the pessimism of the analysis tool, i.e., how close the determined execution time bound is to the real values when executing on the target processor. For the sake of this example, each processor has its own analysis tool, W\textsubscript{A} and W\textsubscript{B}. W\textsubscript{A} generates a worst-case timing bound of 1 100 cycles for the task on processor A, while W\textsubscript{B} generates a bound of 1 300 cycles on processor B. Considering these timing bounds, processor A seems to be able to execute the task faster. But connecting both criteria, the tool W\textsubscript{B} is inferior to W\textsubscript{A} as it generates a tighter execution time bound. Additionally, W\textsubscript{B} might generate a result in one day, while W\textsubscript{A} is able to produce a timing bound in a few minutes, which in turn significantly decreases the value of processor A and its analysis tool in the design process of a system.

As can be seen from this example, Schoeberl argues that predictability alone is an incomplete basis for analysis and consequently, a comparison always has to be drawn using the timing bounds determined by the tools of each processor and system in question and can, if needed, also include factors like power consumption or resource usage.
3.2. Definitions of the term predictability

3.2.3 Relation between worst-case and best-case timing behavior

An extended definition of predictability can also be found in Thiele and Wilhelm [2004] as well as Kirner and Puschner [2011]. As done in Section 2.2.2, Thiele and Wilhelm define a set of parameters essential to the discussion of timing predictability in relation to performance of embedded systems and argue that the differences between the upper and lower bounds on one side and the worst and best cases of execution times on the other are measures for the timing predictability of a system. The closer the determined bounds are to the real values, the more predictable a system is, as illustrated in Figure 3.1. This definition in itself includes a broader approach to predictability than the fact that a system supports it, but is not as wide as the previous definition by Schoebel.

The differences between the determined bounds and the real values stem from two factors: unknown external interference and limited analyzability. For former describes the dependency of the execution time on external influences that cannot be observed by an analysis tool, such as interference from other processors and tasks in a multiprocessor system using a shared communication medium. The latter describes the inability of analysis models to capture all kinds of complex behavior, in hardware as well in software systems. To improve the predictability of systems and mitigate the influences of both factors, they suggest two approaches. The first is to reduce the sensitivity of a system to interference from information not available to analysis tools and the second is to match the implementation of systems with the analysis techniques in use to increase their precision. In the context of hardware development, both influences have been discussed in Section 2.2.1, while the possibilities of mitigation are presented in Section 3.1.

Kirner and Puschner offer a similar definition. They see predictability as a measure of the variability of execution times due to varying input data, external influences as well as variability and interferences inside the system. Therefore, the smaller the interval $[\text{BCET}, \text{WCET}]$, the more predictable a system is. Consequently, the case $\text{BCET} = \text{WCET}$ describes the most predictable system.

This definition of predictability leaves out the analysis tool and performance of a
system but focuses on the degree of variation in the execution times of a task on a given architecture. By eliminating all possible sources of variation, such as hardware features that enhance performance, the width of the interval can be reduced. Additionally, research has been conducted into the field of WCET-aware compilation, which aims to utilize information about the target architecture to reduce the timing variation exhibited by the compiled code [Falk and Lokuciejewski 2010]. However, the execution time variation due to unpredictable input values will still remain. The ideal case stated above can therefore never be reached.
3.2. Definitions of the term predictability
4

Existing Predictable Architectures

While the previous chapter presented the challenges and the requirements connected to the design of time-predictable hardware architectures, the following will introduce current approaches to their creation and how these aim to solve the stated problems.

4.1 CompSOC

One of the largest projects aiming to create a predictable and composable multiprocessor platform template in both hardware and software is the Composable and Predictable Multi-Processor System-on-Chip (CompSOC) project. The following section, based on the work in Hansson, K. Goossens, et al. [2009] and Hansson, Ekerhult, et al. [2011], will present how it addresses and solves the issues and challenges discussed above and how predictability and composability is realized both through hardware and software features.

4.1.1 Goals

CompSOC defines its goals as follows:

i) enable performance guarantees for real-time applications,

ii) allow independent analysis and verification of all applications running on the platform,

iii) support multiple application domains (hard and soft real-time as well as general applications) and

iv) offer run-time reconfigurability.

To enable the first of the listed goals, the possibility to provide performance guarantees, the platform uses components that offer high predictability through formal analysis on the hardware side and a software model that supports the same. To achieve the second item, the possibility to independently analyze and verify the behavior of all applications, the platform is realized as a composable architecture: The behavior of one application does not influence any other task running on the platform. This is realized through a set of features in the interconnection network and all shared memories, which enable
4.1. CompSOC

independent analysis. The third item, the support for multiple application domains, describes the support of not only multiple types of real-time systems running concurrently, but real-time applications as well as tasks without any timing restrictions, so called *mixed-criticality systems*. This is enabled by basing access to the interconnect and shared resources on *enforced budgets*, i.e., assigning a budget of resources and accesses to shared components and enforcing them for each task. The fourth item, the support for run-time reconfigurability is offered through the possibility to dynamically allocate tasks and resources to the system without affecting any other task currently running. Additionally, all hardware components are software-programmable, which means that their properties can be changed from within the system at run-time and do not need to be fixed at design time.

4.1.2 Platform overview

As explained above, the platform employs a range of methods to offer predictability and composability. The following will present both concepts in the context of the platform and introduce the methods used to guarantee them.

Predictability and composability

To offer support for real-time applications, the architecture must provide possibilities to tightly predict the execution time of tasks. To enable predictable sharing of resources, the platform employs hardware-based *resource budget enforcement*, which guarantees a lower bound on resource availability. For the example of the interconnection network, this leads to a guaranteed minimum bandwidth and maximum latency for every task. As explained in Section 2.2.2, this is an integral part in the determination of execution time bounds of tasks in multiprocessor architectures, as without guaranteed bounds, the timing of resource accesses cannot be predicted. To enable this, the platform uses *preemptive arbitration* in all shared resources. A scheduler of this kind interrupts tasks whenever they reach their allocated resources budget, therefore making it impossible for one application to influence any other application on the system and invalidate their predicted execution time bounds. This effectively eliminates all interference between tasks in the system, thereby also enabling a high degree of composability.

Additionally, every component of the architecture supports the determination of tight execution time bounds, as will be explained below.

4.1.3 The hardware platform

The CompSOC hardware platform is built around processor and memory components called *tiles*, which are interconnected by an *NoC*. In addition to that, the system can accommodate input and output tiles as well as a host CPU tile, which is responsible for
Chapter 4. Existing Predictable Architectures

Figure 4.1: The basic CompSOC architecture template showing one processor and one memory tile, the host CPU as well as peripheral components. Additionally, the internal structure of a processor tile is shown, consisting of a MicroBlaze core, interfaces for FIFO streaming ports as well as the NoC and private data and instruction memories. Adapted from [Hansson, K. Goossens, et al. 2009].

the configuration of all components. Figure 4.1 shows the basic structure of the hardware platform.

Interconnection network

The interconnection network used by the platform is the Æthereal NoC presented first in K. Goossens, Dielissen, and Radulescu [2005]. The components shared by the tiles of the platform are the Network Interfaces (NIs) of each tile, the routers, their input and output buffers as well as the network links. Predictability and composability are present on a connection level of the network. These connections realize communication between the NIs of two tiles, where transaction are serialized and broken down into network packets, so called Flow Control Units (Flits). Among the transaction data and other required information, these Flits contain the channel identification of the transaction, describing its source and destination. The arbiters inside the network routers are able to preempt these transactions, if an allocated budget is reached, which enables predictability and composability, as mentioned above. Additionally, the network uses a Time Division Multiplexing (TDM) access scheme to enable a guaranteed minimal throughput for each connection throughout the network and to avoid contention. Figure 4.2 illustrates this behavior. As can be seen there, the TDM tables in each router describe the channel through which communication is allowed in each TDM table slot, while each slot is active only for a certain time, after which the next slot becomes active and communication over its assigned channel becomes possible. Additionally, a slot can be empty or multiple slots can be assigned to the same channel. At the end of the TDM round, it is started again. The example in Figure 4.2 shows two network tiles, IP_A and IP_B, their NIs and two routers. Further, the TDM tables contain four slots, of which two are occupied in NI_A and one is occupied in NI_B. To avoid contention, the slots are occupied in an interleaved fashion. If both slots 0 would be occupied with different channels in each NI, there is a chance that the Flits contend and the composability of the system could not be guaranteed. Throughout the network, all TDM tables have the same size in slots and each slot has the
same fixed size in clock cycles. As a consequence, in each time slot, there is a dedicated connection throughout the network, from the source of a communication channel to its destination with no possibility of disruption or contention.

**ÆElite NoC** To implement global TDM tables and contention-free routing throughout the whole network, the Æthereal NoC requires what is called *global synchronicity*, which means that every router and NI are required to operate at exactly the same frequency and clock phase. As a growing number of network tiles and therefore an increasing chip area make it increasingly difficult to ensure this clock behavior, this requirement presents a strong limitation on the possibility to utilize this routing concept. To mitigate this, Hansson, Subburaman, and K. Goossens [2009] presents a modified version of the Æthereal NoC, called Ælite. This modified NoC structure offers a guaranteed bandwidth as well as maximum latency for each connection between asynchronous actors inside the network, which is achieved by modifying the individual components of the network structure. To allow for a gradual clock skew between each pair of routers, connecting network links are implemented as pipelines that are able to transfer data between these *mesochronous* clock areas. The router architecture contains the largest modifications when compared to the Ætherel NoC. To decrease the resource usage of the network structure, the routers do not contain any arbitration logic. It is therefore necessary for the Ælite NoC that all communication is scheduled offline and the exclusivity of resources is guaranteed outside the network. A similar approach is taken with the interconnection structure developed within the T-Crest project, called S4NoC and presented in Schoeberl et al. [2012]; Sorensen, Schoeberl, and Sparso [2012]. It utilizes simple and lightweight network routers which, however, require static, offline scheduling of all messages to avoid communication failures and to guarantee predictable timing.

---

*Figures and annotations adapted as necessary for clarity.*
Chapter 4. Existing Predictable Architectures

Figure 4.3: A example memory tile in the CompSOC platform, consisting of the memory and its controller, an arbitration module and two shells, each the endpoint of one communication channel over the network. Accesses to the shared resource, the memory controller and its slave port, can be arbitrated in either a TDM or CCSP fashion, depending on the memory technology used. Taken from [Hansson, K. Goossens, et al. 2009].

The modification of the network links is not sufficient to enable communication if neighboring routers are asynchronous. To further allow for connections of this kind, each router is implemented as a stallable process [Carloni, McMillan, and Sangiovanni-Vincentelli 2001], realized by an asynchronous wrapper. This modification stalls each router in its operation if another router or NI at a receiving end is not able to operate fast enough. While this avoids overflowing buffers in the routers and therefore enables safe communication between different clock domains, it reveals the second limitation of the Ælite NoC: The whole network can only operate as fast as its slowest element.

Memory tiles

Throughout the system, memory tiles are shared by multiple applications and multiple processors. As can be seen in Figure 4.3, the shared resource is the slave port of the memory controller. Each shell in the figure is the endpoint of one communication channel across the interconnection network, connected to a remote master port. In turn, this means that every channel to a memory needs a dedicated shell that is responsible for splitting the transactions into Flits before transferring them across the network.

To enable predictable and composable access to the slave port of the memory controller, similar to the NI, accesses are arbitrated in a TDM fashion, with time slots
assigned to the different shells connected to the arbitrator. It is therefore impossible that one master port generates more accesses to the memory than its allocated budget. The TDM tables in each memory tile can be configured independently by the host tile to achieve the timing behavior required by the system and the applications.

To enable the choice and utilization of multiple types of memory at design time, the internal structure of a memory tile is separated in a front end and a back end. The former is comprised of the shells and the NI while the TDM arbitrator, the controller and the memory module itself from the latter. This facilitates that memory accesses can be performed in the same way, regardless of the chosen technology to store the data. However, as mentioned in Section 3.1.2, the utilization of SDRAM presents unique challenges in the context of predictable systems. One solution to these challenges is presented in Akesson, K. Goossens, and Ringhofer [2007] and is used in the CompSOC platform via a SDRAM tile back end.

**Predictable SDRAM controller.** To facilitate a guaranteed minimum throughput and maximum latency to a shared SDRAM, Akesson, K. Goossens, and Ringhofer propose an implementation of a predictable SDRAM controller named *Predictable SDRAM Memory Controller (Predator)*, with two basic functionalities: the first is based on the precise analysis of the timing of single SDRAM operations as well as the internal refresh, activate and precharge operations. Using these, it is possible to create memory access groups that aggregate transactions between the controller and the memory module and are set up in a way that the interference between request is eliminated and refresh and precharge operations happen at known times. Incoming requests to the memory are buffered and reorganized inside the controller until a group can be filled, after which it is executed and the requests are served. If too few requests arrive, the group will be padded with arbitrary data. This enables the prediction of the timing behavior of requests to the SDRAM controller.

To enable composability and avoid interference between multiple requestors, the second functionality is formed by an arbitration mechanism that bounds the latency of memory accesses by scheduling them according to a CCSP scheme, as presented in Cruz [1991]. An arbitrator of this kind grants accesses of requestors based on a priority that is assigned to them and limits their accesses if their allocated budget is reached. This makes this controller feasible in the context of composable systems and integrates with the architecture of the CompSOC memory tiles.

**Processor tiles**

The processor tiles used by the CompSOC platform are based on the MicroBlaze softcore processor, created by Xilinx, Inc. [K. Goossens, Azevedo, et al. 2013; Xilinx 2012]. The cores, shown in Figure 4.1, are customizable and can be adapted to the
requirements of the individual applications running on the platform. In line with the basic requirements stated in Section 2.2.1, the cores implement a range of features to enable high predictability and analyzability to determine tight execution time bounds. First, the cores neither have instruction nor data caches, thereby minimizing the variability in the execution time of instructions and simplifying the creation of an analysis model. Second, the cores do not employ pipeline bypassing or hazard detection, which, in combination with the first property, makes it possible to generate tight execution time bounds for tasks by only considering the compiled machine code for the processor tile. Third, instructions and local data are stored on private memories on each processor tile, as can be seen in Figure 4.1. This eliminates the dependency of the instruction fetch on the state of other applications. Additionally, it enables the timing analysis to separately determine timing bounds for communication across the network and the execution of applications on the processor tiles.

In addition to that, each processing tile is equipped with a dedicated communication memory which forms the only connection to the network. This enables the timing analysis of applications to separate between computation only requiring data stored on the local memory and communication with other tiles connected to the network. To shorten the time required to transfer data between the local memory and the communication memory, a Direct Memory Access (DMA) unit is connected to each MicroBlaze core. To avoid interference between tasks running on a single processor tile, CompSOC does not support SMT on single cores.

As can be seen in this section, the CompSOC hardware platform template offers support for predictability and composability in every component. To ensure analyzability and predictability from the software side, CompSOC enforces a programming model based on abstracting by means of Models of Computation (MoCs). This concept will be further explained in Chapter 5.

4.2 Merasa

Similar to the CompSOC project, the Multi-Core Execution of Hard Real-Time Applications Supporting Analyzability (Merasa) project, presented in Gerdes et al. [2008]; Ungerer et al. [2010] and Paolieri, Mische, et al. [2012], aims to achieve the creation of a multicore platform template supporting both predictability and composability on a hardware level. Furthermore, as the former, it supports mixed-criticality applications, i.e. the simultaneous execution of HRT and Non-Hard Real-Time (NHRT) applications on a single chip while hardware features ensure that interference between those applications is eliminated.

Unlike CompSOC, however, Merasa supports SMT on every core with support for one HRT and three NHRT tasks running concurrently, uses data and instruction caches and
has a bus-based interconnection structure to facilitate communication between cores, shared caches, peripheral devices and off-chip memories. Figure 4.4 gives an overview of the platform and its components and how it achieves predictability and composability.

4.2.1 Predictability

Each core is designed as an in-order SMT processor which prioritizes one HRT over up to three NHRT tasks. To achieve analyzable and predictable timing, the following main features are employed by the cores:

i) an in-order superscalar pipeline which provides high performance and analyzability;

ii) the complete isolation of the HRT task, achieved by granting it the highest priority for every shared pipeline state inside the core;

iii) the use of separate, hardware-managed data and instruction scratchpad memories for the HRT task and traditional caches for the NHRT tasks to avoid interference;
iv) the implementation of long-running operations, e.g. division or multiplication
operations or accesses to external memories, as interruptible microcodes, thereby
avoiding the blocking of shared resources.

4.2.2 Composability

To enable the separate execution time analysis of HRT tasks running on different cores,
Merasa bounds the timing behavior of accesses to resources shared among these cores.
To achieve this, the following features are used:

i) a real-time-capable bus which bounds the delay of transactions

ii) a multibank dynamically partitioned cache, on which data belonging to each HRT
task is kept separate to eliminate interference

iii) a SDRAM controller with predictable timing, i.e. that eliminates unpredictable
delays caused by precharge and refresh operations

4.2.3 The hardware platform

The following will present the individual blocks of the Merasa architecture and how these
implement predictability and composability.

Merasa processing core

The processing cores, described in Mische et al. [2010], are based on the in-order
superscalar TriCore processor developed by Infineon Technologies, with significant
modifications to support multi-threading and the ability to guarantee bounded timing
behavior. As mentioned above, the multi-threading support encompasses four threads,
each owning a dedicated register set inside the core, of which one is a prioritized
HRT thread and the remaining three are NHRT threads. Figure 4.5 shows a single
Merasa processing core and its internal components, all of which will be presented in the
following.

Prioritized instruction fetch and issue. The cores utilize two pipelines, one integer
and one address pipeline, each consisting of five stages. The first two stages, Instruction
Fetch and Real-Time Issue are shared, while the last three, Decode, Execute and Write Back
are separate. Both pipelines can be occupied by instructions from the same task, if an
integer instruction is directly followed by an address manipulation instruction. In every
other case, the pipelines are filled with instructions from different threads.

---

2In the context of Merasa the terms task and thread are used interchangeably
4.2. Merasa

To enable the prioritization of instructions belonging to the \( \text{HRT} \) thread, the threads are scheduled using a \textit{Fixed Priority Preemptive (FPP)} algorithm, i.e., each task has a fixed priority and the task with the highest priority will be pushed into the pipeline, granted that it is ready for execution. The highest priority is always assigned to the \( \text{HRT} \) task, while the remaining tasks can have arbitrary priorities assigned to them.

The real-time issue pipeline stage is responsible for the identification of instructions that can be issued into the pipeline and, as the instruction fetch stage, prioritizes the \( \text{HRT} \) task. In addition to that, it analyzes the instruction stream and determines if a currently running multi-cycle instructions of an \( \text{NHRT} \) thread could possibly stall the execution of the \( \text{HRT} \) task. If this is the case, these instructions are able to be preempted due to their implementation as interruptible microcode sequences. This behavior ensures that no \( \text{NHRT} \) task is able to delay the execution of the \( \text{HRT} \) tasks, thereby generating composability on the \( \text{SMT} \) level.

\textbf{Intra-Core Real-Time Arbiter.} In addition to the ability to interrupt multi-cycle microcode sequences to avoid delaying the \( \text{HRT} \) task, the same must be possible for load and store operations across the on-core bus interconnect, as seen in Figure 4.4. This \textit{Intra-Core Real-Time Arbiter (ICRTA)} issues memory accesses to the local memories, the shared cache as well as off-chip memories in the same prioritized order as the instruction issue pipeline stages issues instruction for execution, which guarantees timing predictability of memory accesses against interference from other threads on the same
While the prioritization of memory accesses scheduled for execution is simple and is handled similar to the prioritized issuing of instruction to the pipelines, running memory accesses cannot be interrupted. To avoid this potential source for interference, the Merasa cores use a technique called Split Phase Load (SPL). A memory access instruction is split into two separate instructions that are scheduled and issues independently: the address calculation and register write back. If a memory access instruction is identified by the instruction issue logic, the address calculation is introduced into the corresponding pipeline and the result is forwarded to the ICRTA. There, the operation is scheduled according to the priority of the task the operations belongs to and performed either on local memories or relayed to the real-time bus system that connects all cores and memories in the whole system. As soon as the operation is completed, the ICRTA notifies the issue stage to schedule the second part of the memory operation, the register write back. As this will be subject to the same prioritized scheduling, composability will be guaranteed throughout the complete memory access.

**Dynamic instruction and data scratchpad memories.** To improve performance and to avoid bus accesses, Merasa uses a two-level cache system. First-level caches are local to each core, while the second-level cache is shared among cores. To eliminate inter-task interference, the first-level caches are kept separate between the HRT and NHRT tasks: traditional instruction and data caches for the latter and dedicated instruction and data scratchpad memories with predictable timing to serve the former. The following will present these scratchpad memories, described in Metzlaff, Uhrig, et al. (2008) and Metzlaff, Guliashvili, et al. (2011), and how they achieve caching characteristics while keeping a predictable timing behavior.

Each HRT task uses two scratchpad memories: a data scratchpad to store stack data and a dynamic instruction scratchpad to store instructions used by the tasks. The functionality of these memories is, unlike traditional caches, not based on the addresses of data, but on its logical characteristics inside the instruction stream. While a traditional cache consecutively loads single instructions, the instruction scratchpad loads whole functions as soon as they are activated by the running thread, i.e. on calling them or returning to them. Figure 4.6 shows the structure and internal components of the instruction scratchpad memory used by the Merasa processor.

If a function is called, its address, calculated by the address pipeline, is transported to the content management unit of the memory, which matches it against function addresses already stored inside the memory. If the function is already loaded and present in the memory, the context management logic loads the internal address and size of the function from the mapping table. This information is then stored in the context register, which holds the current function context. If a function is not present inside the scratchpad memory, the ICRTA is notified and the function is loaded from the data scratchpad.
memory when it is activated, the content management refers a load operation to the [ICRTA] which in turn relays the request to the inter-core real-time bus system. From there, the function will either be loaded from the shared second-level cache or the off-chip memory.

To be able to load a complete function into the memory, information about its size has to be known to the hardware. To accomplish this, [Merasa] makes use of a modification in the software generation tool chain that adds a special hardware instruction to the beginning of every function in the instructions of each task which stores the size of the following function. In order to store each function of a certain application, the instruction scratchpad memories have to be configured to be as large as the largest function of the application to be run on the platform.

**Bus-based interconnect**

The next step in the hierarchy of the [Merasa] platform is the bus-based interconnection structure, described in Paolieri, Quiñones, et al. [2009]. It connects the cores to the shared memories. Requests are issued to this Inter-Core Bus-Arbiter (XCBA) in three situations: a miss in the instruction scratchpad memory of an [HRT] task, an access to the heap memory of an [HRT] task and every data or instruction cache miss of an [NHRT] task.

Out of these types of requests, different timing situations have to be considered in the context of timing analysis of the bus: an [HRT] request and an [NHRT] request arrive at the same cycle, an [HRT] request arrives directly after an [NHRT] request and multiple [HRT] requests arrive at the same time or shortly after each other. The first type is the simplest and is solved by always prioritizing [HRT] requests, similar to the [ICRTA] scheduling policy. For the remaining two situations, however, [Merasa] does not provide a specific solution, but is able to bound to delay that is possible to occur whenever these situations take place.

As all components of the [Merasa] system are predictable, all memory accesses have a
predictable timing behavior, namely the second-level cache and the off-chip memory. It is therefore possible to determine the worst-case delay that multiple HRT task requests can infer on each other, given the execution time of a single bus access and arbitration policy used. If $L_{bus}$ is the former, the arbitration policy is set to round-robin, thereby not favoring a single HRT over another, and $N_{HRT}$ is the number of HRT tasks running in the system, the Upper Bound Delay (UBD) can be calculated as:

$$U BD_{bus} = (N_{HRT} - 1) \cdot L_{bus}$$

As can be seen from that, Merasa does not achieve inter-core composability but is only able to bound the delay inferred by interference of other cores on the bus.

Dynamically partitioned cache

To improve the overall performance of the architecture and to limit the number of accesses to the off-chip memory, Merasa employs a second-level cache that is shared between all cores. However, to eliminate storage interferences, as explained in Section 2.2.1, a cache concept called cache partitioning or bankization is used, described in Paolieri, Quiñones, et al. [2009]. While traditional caches use banks to enable simultaneous accesses to the stored data, the dynamically partitioned cache assigns a set of banks to each HRT task in the system, thereby eliminating the possibility of storage interference and supporting full timing predictability for accesses to this second-level cache. In addition to that, it is possible to dynamically assign banks at run-time, depending on the memory requirements of individual tasks. The integration of the cache is shown in Figure 4.4.

Off-chip memory components

The final memory stage in the Merasa platform is formed by an off-chip SDRAM block and an on-chip controller, called Analyzable Memory Controller (AMC), presented in Paolieri, Quinones, et al. [2009], which offers real-time capabilities by bounding the execution time variation inferred by the internal structure and functionality of SDRAM modules. However, unlike Predator presented in Section 4.1.3, which collects memory requests into groups with predictable timing, AMC performs no reorganization or bundling of requests, but takes advantage of interleaving all possible banks of the SDRAM modules it is connected to, thereby pipelining all memory access to achieve a more predictable timing behavior. This is achieved by analyzing the timing characteristics of the underlying memory modules and performing accesses with a granularity of individual operations on these banks to match the required delay between bank changes. This eliminates timing variations introduced by the activate and precharge command used to open and close SDRAM banks. To accommodate for the additional uncertainty of the refresh command,
4.3 Comparison and summary

In this chapter, two different approaches to solving the challenges of creating a fully timing-predictable and composable hardware architectures were presented. In Table 4.1, these issues are recapitulated and the solutions proposed by CompSOC and Merasa compared in short. As can be seen there, both projects take fundamentally different approaches: While CompSOC does not allow SMT execution and treats all tasks running in the system with the same priority, Merasa allows for multiple tasks on one processor and prioritizes a single thread from this set. In turn, the interconnection structure used by CompSOC, a predictable and composable NoC, inherently allows for multiple parallel connections between network tiles, while Merasa uses a bus-based system, thereby only allowing for a single connection. It is further only able to bound the delay on these bus transactions, while CompSOC offers dedicated communications paths with known delays.

Table 4.1: Comparison of different hardware approaches to minimizing execution time variation.

<table>
<thead>
<tr>
<th>Architectural feature</th>
<th>Standard processor issue</th>
<th>Approach by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline</td>
<td>Dependencies and shared state</td>
<td>Use simple five-stage pipeline</td>
</tr>
<tr>
<td>Caches</td>
<td>Unpredictable replacement, dependency on addresses</td>
<td>Do not use caches</td>
</tr>
<tr>
<td>Branch prediction</td>
<td>Long branch history determines decisions</td>
<td>Do not perform branch prediction</td>
</tr>
<tr>
<td>Superscalar execution</td>
<td>Timing anomalies</td>
<td>Do not perform superscalar execution</td>
</tr>
<tr>
<td>Chip multi-threading</td>
<td>Inter-thread interference in the bus and the pipeline</td>
<td>Do not use SMT</td>
</tr>
<tr>
<td>Chip multiprocessors</td>
<td>Inter-core interference in shared memory, bus and peripherals</td>
<td>Use interference-free TDM routing through the NoC</td>
</tr>
</tbody>
</table>

Merasa performs a platform-wide alignment of the scheduling of HRT tasks with the occurrence of a refresh. As this operation has a fixed interval, aligning the start of tasks and its execution eliminates the uncertainty introduced by it.

Additionally, in line with all components of the Merasa architecture, the memory controller prioritizes HRT tasks to bound the delay of their memory requests.

4.3. Comparison and summary
Using the knowledge gathered in this chapter, the following will present ForSyDe, a system design language and methodology as well as the development of a predictable and composable hardware architecture based on the Altera design tools.
4.3. Comparison and summary
The following chapter will present ForSyDe, a system design language and methodology that aims to solve the problems in the design of embedded systems which emerged with the rise of multiprocessor hardware architectures. The first section will shed light on these challenges and how ForSyDe aims to mitigate them, while the subsequent sections will introduce the basic concepts of ForSyDe and how this thesis integrates with it.

5.1 Introduction

As introduced in the first chapter of this thesis, the design of embedded systems has unique requirements in terms of performance, cost and energy consumption. If the system is to be used in safety-critical environments and its functionality includes real-time requirements, additional care has to be taken to not only fulfill these during the design process but also to prove the timing behavior of the system under all circumstances. While this is a manageable problem for small systems consisting of simple software and hardware architectures, the increase in processor performance and the rise of multiprocessor systems have made it possible to create complex architectures for which traditional testing mechanisms were not designed and therefore can neither make safe predictions about correct functionality nor timing behavior.

To mitigate this problem, it has been suggested to move the design of systems to a higher abstraction level [Keutzer et al. 2000], on which there is the highest level of freedom about the choice of software and hardware implementation as well as about details of basic programming concepts such as concurrency and inter-task communication. Furthermore, creating systems on a high abstraction level enables the application of formal methods to automate important tasks that can currently only be performed manually by the designer: the exploration of the possible design choices and their effect on the final implementation of the system, analysis of the performance of the system, verification of both functionality and timing as well as the automated creation of specific implementation, i.e., the automated bridging of abstraction gaps.

This idea forms the concept behind ForSyDe developed by Sander et. al. [Sander
5.1. Introduction

Using ForSyDe systems are designed as hierarchical networks of concurrent processes, each confirming to a MoC which form the basic language of the design system. This specification model describes the functionality of the system, i.e., what the system is supposed to do, rather than how the system should achieve this functionality. Next to this basic language, in which systems are described, ForSyDe as a concept is based on two more important parts: the modeling framework, which implements the basic language and defines the syntax to create process networks, and the design methodology, which encompasses the possible applications of the ForSyDe process model to, e.g., perform design decisions, which gradually lower the abstraction of the model down to implementation model, on which it can be compiled for a system or synthesized into hardware. These decisions can either be based on code transformations or a refinement-by-replacement strategy and the application of mathematical methods, such as those employed by a number of Design Space Exploration (DSE) tools [Schätz, Hözl, and Lundkvist 2009]. While code transformations are out of the scope of this thesis and will not be introduced further, the possibility to make use of mathematical analysis of the ForSyDe process model will be explained.

5.1.1 Models of Computation

As mentioned above, the analysis of the timing behavior of individual processes and process networks is an important part of the design process of real-time embedded systems. To introduce the concept of computation and communication timing, ForSyDe utilizes Models of Computation (MoCs). These models specify the semantics of computation and communication while staying on a high abstraction level. Currently, the ForSyDe language supports several MoCs, which each define a set of process constructors of the three categories stated above. In addition to that, it is possible for processes belonging to...
different MoCs to communicate with each other using Domain Interfaces (DIs), which can also be seen in Figure 5.1. Out of the group of supported MoCs two will be introduced in the following.

5.1.2 Synchronous Model of Computation

The synchronous MoC is a model based on the notion of perfect synchrony [Benveniste and Berry 1991; Herrera and Villar 2008], in which processes are infinitely fast and the propagation of signals takes no time, i.e., the outputs and inputs of a system are synchronized. In consequence, the outputs of processes are available immediately on arrival of their input signals. This eliminates the possibility of race conditions between processes and thus simplifies the analysis of network based on this model. This timing behavior is also the basis of the Esterel programming language [Berry 1999].

5.1.3 Synchronous Data Flow

The SDF MoC, first described in E. Lee and Messerschmitt [1987], defines additional semantics for the input and output signals of processes. Under SDF, processes can start executing, or firing, as soon as a certain amount of input data tokens has arrived. Similarly, every process creates a certain number of output tokens during each execution cycle. It can therefore be said that during each firing, a process consumes and produces a certain number of tokens. Figure 5.2 illustrates this in example SDF process network.

The reason for adding this additional information to the process network lies in the possibility to create static schedules that allow the execution of the process network only considering the number of tokens consumed and produced by each process. These guaranteed periodic admissible schedules can be calculated statically using the number of tokens and have the property that after each schedule round, all processes have fired at least once, the number of tokens in the system returned to its initial state and a new round begins. This repeats indefinitely. Additionally, the creation of these schedules
5.2 ForSyDe processes

Using the example of the Synchronous and SDF MoC, ForSyDe offers a number of basic building blocks to describe the functionality of a system. The set of processes in these two MoCs consists of two categories: combinational and sequential, with the addition of delays, a special case of sequential processes. While each process takes $m$ input signals as an argument and produces $n$ output signals, the two categories differ in their timing behavior. The first category, combinational processes do not have an internal state but perform operations on their input signals in no observable time and therefore have a behavior that can be compared to that of combinatorial functions in hardware. The second category, sequential processes form a combination of combinational processes.

Figure 5.3: A very simple example of an SDF process network using two processes. Process A produces two tokens while process B consumes three. Two correct schedules are shown, which, however, differ in the required buffer sizes for process B. The first schedule fires process A three time before process B gets executed. Using this schedule, process B needs to hold six tokens in its input buffer. The second schedule, however, fires process A twice before process B executes and consumes three tokens, thereby decreasing the required input buffer size to four.

enables the identification of the channel buffer sizes required to hold produced tokens, called static buffering [E. Lee and Messerschmitt 1987]. Figure 5.3 illustrates this in a very simple example using two processes: For process B to execute, process A needs to fire at least twice. To store this amount of tokens, the buffer size for process B must be at least 4 and can, if another execution schedule is chosen, also increase to 6. As can be seen from this, although different schedules exist, they place different restrictions on the buffer sizes of the processes and therefore on the cost of the implemented system.

Homogeneous Synchronous Data Flow. A special case of SDF process networks is the one in which each process consumes a single token at its input and produces a single token at its output, i.e., all annotation in the network are 1. These networks are called Homogeneous Synchronous Data Flow (HSDF) networks. E. Lee and Messerschmitt [ibid.] and E. A. Lee [1986] show that it is possible to convert every SDF process network into an HSDF network. If necessary, nodes can be duplicated to enable this conversion, as shown in Figure 5.4. This is possible due to the fact that all processes are combinational and side-effect-free, therefore produce the same output token for each input.
and delay elements. These processes contain an internal state, perform operations on input signals and delay their output. This mirrors the functionality of state machines, in hardware realized as combinational functions and registers to store the internal state.

Processes of all two categories are designed by means of process constructors, which defines the communication interface of the process with the process network. These constructors combine a side-effect-free function and variables into a process. Figure 5.5 illustrates this behavior with the creation of a sequential process using a process constructor while Figure 5.1 shows an example of a ForSyDe process network. The concept of functions being side-effect-free is an integral part of the idea behind ForSyDe: if a function does not depend on any other variables except its input signals, it is possible to not only gradually replace its functionality during the design process but also to separate computation and communication, which makes formal verification of both functionality and timing of the process network possible. This possibility is strengthened by the inherent property of ForSyDe process networks: they only consists of processes which each adhere to one of three categories listed above and thereby force the designer to develop a structured model of the system.

5.3 Modeling framework

To be able to use the ForSyDe language basis to create system, it has been implemented in two distinctly different programming languages. Originally, it was implemented as a library in the functional programming language Haskell, as there are a number similarities
between the programming concept behind Haskell and ForSyDe, such as the theory of side-effect-free functions and the usage of higher-order functions to create processes. This thesis, however, will focus on the second implementation based on SystemC.

### 5.3.1 ForSyDe-SystemC

ForSyDe-SystemC is an Embedded Domain-specific Language (DSL), developed as a template class library on top of SystemC [Attarzadeh Niaki, Jakobsen, et al. 2012]. Unlike Haskell, whose functional programming concepts more closely resemble the theory behind ForSyDe, SystemC, and therefore ForSyDe-SystemC, is based on C++, an object-oriented programming language. A wide range of language constructs and possible programming techniques therefore have to be avoided to create pure ForSyDe process networks that retain their formalism and analyzability. For example: While functions in Haskell are always side-effect-free, functions in C-based languages allow side effects via global variables, a commonly used concept.

All elements of ForSyDe-SystemC are implemented as classes inside the ForSyDe namespace. Additionally, a sub namespace is defined for each MoC which in turn contains all elements belonging to this MoC. For example, the namespace ForSyDe::SDF contains all process constructors, signals and DIs belonging to the SDF MoC.

### Signals

Signals are implemented as template classes and connect to the inputs and outputs of ForSyDe processes. Each MoC defines its own class to send and receive tokens across the process network, which is typed with the data structure of the token itself. Listing 5.1 shows an example of the instantiation of a signal to connect two SDF processes, carrying tokens of type int.
Chapter 5. System Design using ForSyDe

Listing 5.1: A ForSyDe signal example

```cpp
ForSyDe::SDF::SDF2SDF<int> sdf_signal;
```

Listing 5.2: The definition of a ForSyDe SDF function that increments the input int token by 1. Additionally, the use of `pragma` compiler instructions is shown, which are used to mark the beginning and end of the functionality of the ForSyDe process.

```cpp
void increment_func ( std::vector<int> & out1 , const std::vector<int>& inp1 )
{
    #pragma ForSyDe begin increment_func
    out1[0] = inp1[0] + 1;
    #pragma ForSyDe end
}
```

Processes

Processes are created using process constructors, which are implemented as helper functions to create these leaf processes, or by combining a set of processes into a composite process. Each MoC defines a group of helper functions to create combinational, sequential and delay-type leaf processes. As Figure 5.5 shows, one or multiple function objects as well as variables have to be provided, which are combined into an instance of the SC_MODULE class that acts as a ForSyDe process. Listing 5.2 shows the definition of a function while Listing 5.3 shows the creation of a combinational process belonging to the SDF MoC using this function. Additionally, process constructors are defined in multiple variations to accommodate a wider range of applications. For example, it is possible to define combinational processes with two, three and four input signals and different kinds of sequential processes, such as Moore and Mealy machines.

Listing 5.3: A ForSyDe process constructor wrapping the increment_func function in a process. The input and output rates are both defined as 1. Additionally, source and sink processes are defined to generate a sequence of numbers and print them to the console.

```cpp
SC_MODULE(top)
{
    SDF2SDF<int> incr_input , incr_output ;

    SCCTOR(top)
    {
        make_source("stimuli_1", stimuli_func, 0.0, 0, incr_input);
        make_comb("increment_1", increment_func, 1, 1, incr_output, incr_input);
        make_sink("report_1", report_func, incr_output);
    }
};
```

The second kind of processes, composite processes, implement the hierarchical component of the ForSyDe process network. Using these processes, it is possible to combine
5.4 Design methodology

a set of leaf processes and let them appear as a larger unit. To accomplish this, an 
\texttt{SC\_MODULE} is extended by input and output signals, as can be seen in Listing 5.4.

```
#include <forsyde.hpp>
SC_MODULE(top) {
  SDF_in<int> comp_input_1, comp_input_2;
  SDF2SDF<int> incr_to_add;
  SDF_out<int> comp_output;

  SC_CTOR(top) {
    make_comb("increment_1", increment_func, 1, 1, incr_to_add, comp_input_1);
    make_comb2("adder_1", adder_func, 1, 1, 1, comp_output, incr_to_add, 
                comp_input_2);
  }
};
```

**Listing 5.4:** A \texttt{ForSyDe} composite process wrapping two leaf processes. The increment\_func function and the adder function are connected and the added values form the output of the composite process. Additionally, a second process constructor make\_comb2 is shown, which creates a sequential process with two inputs and one output.

**XML representation**

The \texttt{ForSyDe\_SystemC} implementation is able to extract structural information about the created process network by means of an \textit{introspection} feature. If it is enabled, a simulation of the model will generate an \textit{Extensible Markup Language (XML)} model of the process network, which contains all processes and signals as well as their data types. In addition to that, MoC-specific information will be included in the representation, such as the number of input tokens for each process in the SDF MoC. Listing 5.5 shows an example of an XML representation of the composite process shown above.

5.4 Design methodology

As mentioned above, basing the design of embedded systems on formal methods offers a range of applications which are either impossible to achieve with traditional system design, or only with considerable effort and care. For example, a refinement-by-replacement strategy, which gradually replaces components of the system while keeping the surrounding parts untouched, is entirely possible when using a formalized process network, such as \texttt{ForSyDe} does [Sander 2003]. A traditional system, however, in which individual components have interdependencies in either timing or functionality, does not allow for this design strategy and the replacement of components, e.g., the exchanging of a software filter with a hardware-based implementation or a processor with a faster model, cannot be performed automatically but need the attention of the developer.
Chapter 5. System Design using ForSyDe

Listing 5.5: The XML representation of a ForSyDe process network consisting of three SDF processes.

Another possibility to reach an implementation model from the ForSyDe specification model is based on the utilization of hardware platform templates and is illustrated in Figure 5.6. A hardware platform template forms the basis of the hardware implementation, while a ForSyDe model describes the software to be run on the hardware. The platform makes it possible to utilize an arbitrary number of processing elements, define shared and private memory sizes as well as the communication infrastructure that is required to house the application. The following will present the individual parts of the ForSyDe design flow as well as the steps towards the implementation level. Bamakhrama et al. [2012] presents a similar flow, while Teich [2012] gives an overview of current approaches to the field of hardware/software codesign, i.e., the concurrent design of software and hardware components.
5.4. Design methodology

5.4.1 Initial flow components

There are three inputs to the ForSyDe design flow: the specification of an executable model of the functionality as a ForSyDe process network, a hardware architecture template and a set of design constraints on the system, the design process and the final implementation.

Executable functional specification

The initial step is the creation of an executable and analyzable specification model of the application to be implemented on the system. As the ForSyDe-SystemC model has a syntax overhead, e.g., the unpacking and packing of tokens into their respective signal classes, the functions to be implemented on the platform need to be surrounded by the aforementioned pragma instructions indicating the begin of the actual functionality of the process.

Design constraints

The design constraints are a set of limiting parameters on the design flow itself and the final implementation. Possible constraints on the final implementation can be to
reach a desired timing behavior or to fall below a certain power consumption or resource limit. In the design of real-time embedded systems, it is always a governing constraint to reach a desired execution time for individual processes and the whole process network. Additionally, the design constraints can describe restrictions of the target platform, such as limited amounts of memory, delays for communication and computation and implementation costs.

**Hardware platform template**

The hardware platform template forms the basis of the hardware implementation of the specification model. As mentioned above, a platform template does not describe a specific system with a set of processing elements, memories and an interconnection structure, but a multiprocessor architecture, customizable in each of these elements. CompSOC and Merasa are examples of platform templates and this thesis aims to provide its own customizable architecture and integrate it into the ForSyDe design flow. As the current design flow is focused on the creation of hard real-time embedded systems, the platform template needs to offer timing predictability and composability. As the example of CompSOC in Section 4.1 showed, a platform template makes it possible to use an arbitrary number of processing elements, memories and peripheral components while the architecture itself ensures composability and offers predictability. This enables a separation of concerns: The application is formalized by the specification model, hardware and software are limited by the design constraints and the application itself is not required to implement features ensuring composability.

### 5.4.2 Design flow steps

Taking into account the three individual parts listed above, the design flow aims to automatically generate an implementation of the ForSyDe model while satisfying all design requirements and constraints.

**Design Space Exploration**

The Design Space Exploration phase of the design flow forms the most important and computationally expensive part of the ForSyDe design flow. As the name suggests, it is responsible for analyzing the range of specific hardware implementations which are able to house the developed specification model. Additionally, valid implementations are required to satisfy the design constraints listed for the application in questions, such as that the execution time of the application on the platform needs to fulfill the real-time requirements or that the cost of the platform needs to fulfill the cost requirement. These implementations differ in the mapping of processes to processing elements in the
5.4. Design methodology

Figure 5.7: Three possible DSE tool mapping solutions, each differing in resource usage and performance.

hardware platform, i.e., which process of the ForSyDe process network is run on which hardware processor.

Figure 5.7 illustrates three possible mapping situations suggested by a DSE tool. A simple ForSyDe process network consisting of three processes is analyzed for implementation on a platform template and while each mapping is valid and is able to execute the process network, they infer different levels of resource usage and power consumption as well as different timing behaviors. The leftmost solution offers the lowest cost as it only uses a single processing element but sacrifices performance to achieve this goal. The rightmost solution offers the highest performance as it uses a one processing element for each process, thereby minimizing the execution time of a single round of the process network: While process C is processing the data received from processes A and B, process A is able to start generating data to be sent to B and C. However, the cost of this implementation is the highest, as it uses the maximum number of processing elements. The second solution, shown in the middle, compromises between execution time and implementation cost by utilizing two processing elements and making use of the inherent parallelism offered by the process network. It is the task of the DSE phase of the design flow to identify these possible implementation models, check their validity against the design constraints and find the ideal model yielding maximum performance while requiring the least amount of resources. While this is a manageable problem for a very small number of tasks, larger process networks in combination with a set of design constraints easily create a design space too complex to fully analyze in a reasonable amount of time [Kang, Jackson, and W. Schulte 2011]. Therefore, DSE tools are topic of
a large number of research endeavors with the goal to identify algorithms that are able to more efficiently cover large design spaces.

An additional task of the DSE phase in the ForSyDe design flow, which further adds to its complexity, is the creation of schedules for both communication and computation. As the underlying hardware architecture provides composability and predictability, it is possible to extract WCET bounds for each process and data transfer in the SDF network. When reconsidering the previous example in Figure 5.7, the rightmost solution is expected to show the highest performance as all tasks can run concurrently as soon as their input data arrives. This, however, does neither take into account the timing overhead needed to send this data between the processing elements nor the WCET bounds predicted for each task. Assuming the delay inferred by sending data to and from task B is larger than the execution time of task B itself, this mapping solution shows lower performance than initially concluded. In fact, under these circumstances, solution 3 will perform worse than solution 2 and use more resources. When creating a valid schedule and a mapping, the DSE tool is responsible for identifying these situations.

**Fully-static schedules.** The creation of a schedule of computation and communication is based on the WCET and Worst-Case Communication Time (WCCT) of the network input into the DSE tool. If the exact execution times of all processes are known, it is possible to create a fully-static schedule, in which each operation in every clock cycle is predetermined during the DSE phase [Sriram and Bhattacharyya 2009]. If this behavior can be guaranteed, the created schedule will not be violated during the execution on the hardware platform and no mechanism to either correct process timing or check the validity of data being sent between the processes is necessary. However, as stated in Section 3.2.3, embedded systems react to inputs from their environment, which is inherently non-deterministic and a perfect prediction of timing behavior can therefore never be given at design time. This uncertainty, i.e., the interval between BCET and WCET, can invalidate a fully-static schedule and lead to errors during the execution of the process network, such as overflowing input buffers for processes in case a sending process executes faster than predicted.

**Self-timed schedules.** In the case described above, during which the execution times of processes cannot be determined but only bounded at design time, the underlying hardware needs to provide support to synchronize the execution of individual processes and communication with the created schedule. These self-timed schedules [E. A. Lee and Ha 1989] can be created using the order of processes implied by the SDF process network. However, a run-time flow-control mechanism needs to be present to check compliance with this statically determined execution order. One such mechanism could be realized using FIFO buffers between processes that store input tokens. If the buffer is full, the
5.5 Conclusion

The final stage of the ForSyDe design flow is the generation of executable code for the target architecture. This step takes place after the completion of the DSE phase, as the specific architecture layout and the process mapping need to be known. Using this information, the design flow inserts the communication mechanisms into the code and compiles the software. At the same time, the hardware platform is generated and synthesized for implementation on the target hardware, an FPGA.

5.5 Conclusion

This chapter introduced ForSyDe, the embedded systems design language and modeling framework based on the theory of Models of Computation. Further, the design flow from a ForSyDe process network down to an MPSoC instance executing it while confirming to execution time bounds and offering predictability and composability was introduced. The following will present the Altera system design tools and a hardware architecture template offering these capabilities based on it.
6 Development & Implementation

The following chapter will present the implementation of a multiprocessor architecture template offering predictability and composability and how it achieves these features in every component. The architecture is based on the Altera system development tool SOPC Builder and Nios II soft processors, which will be introduced in the first part of this chapter. The second part will explain how predictability and composability can be achieved by modifying selected components of the system. In the third part, the individual steps required to create hardware and software of a multiprocessor system will be explained as well as the contribution of this thesis to automate these steps and integrate them into the ForSyDe design flow.

6.1 Altera design tools

In the context of this thesis, three components of the Altera suite of embedded systems design tools are used:

- **Quartus II**, an FPGA design and synthesis software [Altera 2012b];
- **System-on-a-Programmable-Chip (SOPC) Builder**, a part of the Quartus II design suite, which enables the creation of multiprocessor systems using a graphical interface [Altera 2010], encapsulating a range of predefined hardware components, such as Nios II processors, memory controllers and input and output facilities. The most important task of SOPC Builder is to automatically generate the Avalon Switch Fabric interconnection structure, which connects all instantiated components and enables connected processor to access them;
- **Nios II IDE**, a software development environment enabling the automated creation of required Hardware Abstraction Layers (HALs) for Nios II-based systems [Altera 2011b], including all communication facilities and drivers.

In the following, the individual components of a Nios II-based multiprocessor architecture, created using SOPC Builder, will be presented. Further, the amount of timing variation they introduce will be examined, as discussed in Chapter 2.
6.1. Altera design tools

Figure 6.1: The Nios II processor architecture showing elements common to all processor types in gray and optional parts in white.

6.1.1 Nios II processors

Nios II processors are a family of 32 bit soft processors developed for implementation on Altera FPGAs. They offer three different configurations, which share a basic architecture layout and instruction set, which can be seen in Figure 6.1.

- **Fast (/f core)**: the fastest variant, offering the largest number of features to increase average performance while occupying the highest amount of FPGA resources;
- **Standard (/s core)**: the standard configuration, requiring a smaller number of resources than the fast core by sacrificing extended features such as dynamic branch prediction and data caches;
- **Economy (/e core)**: the smallest-sized core, using the least amount of FPGA resources.

The following will further introduce each processor configuration and investigate their feasibility for hard real-time systems requiring support for the determination of tight execution time bounds.

**Nios II/f**

The largest core configuration, the Nios II/f, utilizes features such as dynamic branch prediction, caching for both instructions as well as data, a six-stage pipeline for instruction execution and hardware multiplication and division units. While these features increase the average performance of the processor, their application introduces execution time
variations and dependencies between instructions which lower the execution time analyzability of the processor.

**Six-stage pipeline.** The instruction pipeline of a Nios II/f core is divided into six stages: Fetch, Decode, Execute, Memory Access, Align, Writeback. In the design of the pipeline, two stages are able to stall the complete pipeline and even flush it, if necessary: the Decode stage, in case a delayed result is needed by the instruction currently being decoded and the Align stage, every time a multi-cycle instruction is executed or a bus access is performed. Consequently, the execution time of subsequent instructions is increased. These dependencies between instructions, as described in J. Hennessy and D. Patterson [2011, p. 148], lead to a severe complications in the creation of an analysis model.

**Data and instruction caches.** For both instruction fetches and data accesses the Nios II/f core uses direct-mapped cache implementations. While the instruction cache has a fixed structure, the data cache is configurable in line size as well as addressing scheme. As explained in Chapter 2 and Lim et al. [1995] as well as Reineke [2009], both caches introduce the possibility of timing anomalies into the system as well as strong interdependencies between instruction accessing data via the cache. In combination with the unknown latency of cache misses and replacement operations, it is impossible to tightly model the impact of these caches on the WCET analysis of Nios II/f processor cores.

**Dynamic branch prediction.** To accelerate the execution of branches, Nios II/f cores utilize a dynamic branch prediction scheme. As explained in Section 2.2.1, such a system introduces timing variation into the execution of branches as their prediction is dependent on the branch history. A predicted taken branch, for example, infers a latency of two clock cycles, while a predicted not taken branch can be executed in a single clock cycle. Every mispredicted branch requires a flush of the pipeline and therefore increases the execution time of the particular branch to four clock cycles. In addition to that, the precise prediction scheme is not disclosed by Altera, which makes the creation of a simulation or analysis model impossible.

**Hardware-accelerated multiplication and division.** A Nios II/f core can be configured to utilize different hardware-based multipliers and dividers. While these enable the execution of multiplication and division instructions in a fixed number of clock cycles, the design of the pipeline and the connection to these specialized units introduce a delay of two clock cycles before the result is written back into the register set and becomes available to the following instruction. Listing 6.1 illustrates the impact of this behavior.
In this example, the content of registers \( r_2 \) and \( r_3 \) is multiplied by a dedicated multiplier with a latency of one clock cycle and written back into register \( r_3 \). Since the second instruction, the adding of the immediate value 100 to register \( r_1 \), depends on the result of the multiplication operation, the processor has to be stalled for two clock cycles. As can be seen in Listing 6.2, two single-cycle or operations without data dependencies between the multiplication and the adding enable a non-stalling execution of the instruction stream. As can be seen from this example, while hardware-based multipliers increase the average performance of the processor, they also introduce interdependencies between instructions that severely complicate the determination of tight execution time bounds.

```
1 mul r1, r2, r3 ; r1 = r2 * r3
2 addi r1, r1, 100 ; r1 = r1 + 100 (Depends on result of mul)
```

Listing 6.1: Stalling multiplication on a Nios II/f core

```
1 mul r1, r2, r3 ; r1 = r2 * r3
2 or r5, r5, r6 ; No dependency on previous results
3 or r7, r7, r8 ; No dependency on previous results
4 addi r1, r1, 100 ; r1 = r1 + 100 (Depends on result of mul)
```

Listing 6.2: Non-stalling multiplication on a Nios II/f core

Nios II/s

Compared to the fastest core, the Nios II/s uses approximately 20% less FPGA resources while sacrificing 40% of its performance. In doing so, the core is able to eliminate a range of features that introduce a large amount of timing variation, as discussed in the previous section. However, not all sources of unpredictability are removed, which the following will outline.

**Five-stage pipeline.** Similar to the Nios II/f core, the standard core uses pipelined instruction execution to increase average performance. The pipeline is divided into five stages: Fetch, Decode, Execute, Memory and Writeback. Multi-cycle instructions cause the pipeline to stall, and cache as well as branch prediction misses cause a flush of the whole pipeline. While the delay inferred in these cases is reduced by one clock cycle due to the shortened pipeline, these interdependencies between instructions complicate a tight WCET analysis of applications running on a Nios II/s core.

**Static branch prediction.** A Nios II/s core utilizes static branch prediction to decrease the average delay caused by branches in the instruction stream. The prediction scheme, described in Altera [2011a], uses the direction of the branch offset to predict a branch as taken or not taken. As discussed in Chen, Malik, and August [2001], it is entirely possible
to integrate a simple scheme of this kind into an WCET analysis model. However, the possibility of timing anomalies in combination with the instruction cache is still present and complicates the use of such a branch predictor in hard real-time embedded systems.

**Instruction cache.** Similar to the Nios II/f core, the standard core uses a direct-mapped instruction cache to store blocks of instructions which suffers from the same problems as discussed above and in Chapter 2.

**Hardware-accelerated multiplication and division.** The same timing variations inferred by the choice of hardware-accelerated multipliers and dividers in combination with the instruction pipeline that are apparent in the Nios II/f core, are present in the standard core. The delayed availability of the results of these operations introduces dependencies between instructions that complicate the creation of an analysis model supporting tight WCET determination.

**Nios II/e**

The smallest core of the Nios II family, the Nios II/e, occupies approximately 35% of the FPGA resources of the fast core. This is achieved by removing all extended features while keeping compatibility with the Nios II instruction set. Due to this, a Nios II/e core does neither utilize instruction or data caching nor performs branch prediction of any kind. In addition to that, it does not pipeline the execution of instruction. Consequently, the processor fetches and executes a single instruction at a time and there is no interdependency introduced by the architecture. The Nios II/e core does therefore not suffer from any of the sources of timing variation listed above, making it a fully predictable processor architecture. All instructions are executed in a fixed amount of clock cycles as shown in Table 6.1. The only instruction types that do not confirm to this are all shift and rotate commands. Their execution time, due to the absence of an accelerating hardware feature such as a barrel shifter, depends on the value of the shift width, i.e., on the amount the operand is to be shifted. While a shift by an immediate, fixed amount can be simply translated by an analysis tool, a shift by a dynamic value has to be considered to have the largest delay of 38 clock cycles.

**Software-emulated multiplication and division.** Unlike the standard and fast cores, the Nios II/e can not be configured to use any dedicated hardware to perform multiplication or division operations. These instructions therefore have to be emulated in software and are inserted automatically by the software compilation process for a Nios II/e core. Due to this, the execution time for both multiplication and division operations strongly depends on the operands, as the software emulation uses loops and shifts as well as add and subtract operations, respectively, to achieve multiplication or division of the
6.1. Altera design tools

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU operations</td>
<td>6</td>
</tr>
<tr>
<td>Branches</td>
<td>6</td>
</tr>
<tr>
<td>Load word</td>
<td>6</td>
</tr>
<tr>
<td>Load halfword</td>
<td>9</td>
</tr>
<tr>
<td>Load byte</td>
<td>10</td>
</tr>
<tr>
<td>Store</td>
<td>6</td>
</tr>
<tr>
<td>Shift &amp; rotate</td>
<td>7-38</td>
</tr>
<tr>
<td>All other instructions</td>
<td>6</td>
</tr>
</tbody>
</table>

*Table 6.1: Execution time of instructions on a Nios II/e core.*

operands. The precise mechanisms are documented in the software generation flow and, as their individual parts are fully predictable, their operation as a whole is predictable and fixed for each set of operands.

6.1.2 Memory elements

SOPC Builder-generated systems are able to use any kind of memory element whose controller conforms to the interface specifications set by the Avalon Switch Fabric, described in Altera [2013a]. As explained in Section 3.1.2, the choice of memory technology has a strong influence on the amount of timing unpredictability that is introduced into the system. While SDRAM suffers from a high degree of unpredictability, SRAM and on-FPGA memories perform accesses in fixed amounts of clock cycles without timing dependencies between individual accesses, making them fully predictable. In the context of this thesis, SDRAM will be excluded due to the difficulties connected with its application in hard real-time embedded systems.

SRAM controller. The SRAM controller utilized by this thesis is described in Altera [2012c], while Figures 6.2 and 6.3 show its timing behavior during a read and write operation, respectively. As can be seen there, a read operation of a complete 32 bit word infers a delay of five clock cycles while a write operation of a word infers a delay of two clock cycles. As there are no dependencies between individual operations, these access patterns are always guaranteed. Together with the interconnection protocol overhead, the access times to SRAM components of this kind are therefore fully predictable and can be easily utilized in the static determination of BCET and WCET bounds.

On-chip memory blocks. The memory technology with the lowest access delay is formed by on-chip memory blocks which are embedded on the FPGA [Altera 2013b, p. 37]. Their operation can be highly customized as to their data width and depth, clock
behavior and the number of concurrent read and write ports. The configuration chosen is that of single-port memory blocks with a data width of 32 bit and unregistered output signals. This ensures a read latency of a single clock cycle for a whole word, i.e., after applying the address at a rising edge of the memory clock, the data is available on the next rising edge. A write operation is performed without any delay, as an address and data word can be applied to the memory blocks on a rising edge and are registered internally. The next write operation can commence immediately with the next rising clock edge. Figure 6.4 shows this behavior using the example of four consecutive accesses to the memory: two read operations followed by two write operations.

Similar to SRAM blocks, the on-chip memory elements do not introduce dependencies between individual accesses and can therefore be considered predictable in terms of their timing behavior. In addition to that, their utilization to store instruction and data for a single Nios II processor renders the usage of caches redundant, as it is not possible to achieve an increase in access speed.

Figure 6.2: The timing of two write accesses to the SRAM controller. The first write to address 0x00000000 writes 0xAAABB and receives an acknowledgment from the controller in the next clock cycle, after which the next address and data word can be applied for writing. The delay to write a single word is therefore two clock cycles.

Figure 6.3: The timing of a read access to the SRAM controller used by this thesis. Since the data inside the memory is registered through sram_dq and saved as 16 bit half words, the delay to read a complete 32 bit word is five clock cycles. As soon as the data is ready, the controller signals raises the acknowledgment signal and the output register read_data can be read.
6.1. Altera design tools

Figure 6.4: The timing of read and write accesses to on-chip memory blocks used by this thesis. Due to the asynchronous outputs, the result of a read operation is present one clock cycle after applying the address while a write operation does not cause any external delay.

6.1.3 Interconnection structure

The structure connecting all processors, memories and peripherals created by SOPC Builder is called Avalon Switch Fabric [Altera 2006a, 2013a] and forms its own category of interconnection structure, sharing properties of bus-based interconnection systems as well as of NoCs. Its main characteristic, displayed in Figure 6.7, is the slave-side arbitration scheme which consists of a dedicated arbitration mechanism for each slave in the system. Using this, it is possible for multiple masters to simultaneously communicate with different slaves and arbitration is only necessary as soon as multiple masters aim to access the same resource. This differentiates the Avalon Switch Fabric from a traditional bus interconnection structure, in which only a single master can communicate at any given time. When considering the composability of such a structure, it becomes apparent that applications sharing only the interconnection structure but no other resources do not interfere with each other, thereby guaranteeing composability under such circumstances and marking a difference to traditional bus-based systems as well as NoCs. However, when multiple masters in the systems share not only the interconnection structure but slave memories and peripherals, composability needs to be ensured by the arbitration scheme of the shared slave module arbiter. The following will describe the standard arbitration scheme employed by the Avalon Switch Fabric, its shortcomings in the context of composability and an arbitrator designed to replace the standard scheme, which aims to offer composable sharing of slaves in the Avalon Switch Fabric and therefore predictable access times to shared resources.

Typical read and write transfers

Figure 6.5 shows the timing behavior of typical read and write transfers over the Avalon Switch Fabric for slave modules using a waitrequest signal to stall the requesting master until they are able to process the incoming request. The exact amount of delay inferred by the usage of the waitrequest signal is not defined by the interconnection structure but by each slave itself. For the example of a read operation from an SRAM slave module, the waitrequest signal is asserted for a duration of four clock cycles before the data
Figure 6.5: Typical timing of read and write operations of the Avalon Switch Fabric using slave_wait signals to stall master requests until the called slave module is able to finish the requested operation.

is available and can be read by the requesting master. As can be seen from both this example and Figure 6.5, the interconnection structure itself does, due to its structure, not add any delay to the execution time of requests as signals. This is due to the fact that signals are not registered by the arbitration module but merely pass through a multiplexer into the slave module.

**Slave-side arbitration mechanism**

SOPC Builder automatically generates an arbitrator for every slave port connected to multiple master ports, as can be seen in Figure 6.7. The functions this module performs are those of an arbiter used in a traditional bus:

- evaluate the address and control signals from each master port and determine if any master port is requesting access to the slave;
- choose which of the requesting masters gains access to the slave during the next clock cycle;
- grant access to the correct master and generate wait signals to signal those masters that are not currently granted access (via the master_n_waitrequest signal);
- use multiplexers to connect all signals of the appropriate master port to those of the slave port.

To decide which master is granted access in which clock cycle, the arbitrator uses a fairness-based round-robin arbitration scheme with support for variable amounts of shares for each master. Its operation is shown in Figure 6.6. In this example, two masters are connected to one slave port, to which both continuously request access. Master 1 is assigned three arbitration shares, i.e., it is allowed to perform at most three operations in a row, after which Master 2 gains access to perform at most four transfers. As both are continuously requesting access to the slave, the arbitrator, due to its round-robin mode of
6.1. Altera design tools

Figure 6.6: The timing of the arbitration of continuous concurrent master port requests to a single slave port.

operation, grants access to Master 1 for three transfers, then signals Master 1 to wait and grants access to Master 2 for four transfers, after which the cycle repeats.

Figure 6.8 shows the behavior of the arbitrator in the case of an interrupted request during the operation of one master. In this example, Master 2 has been granted access to the slave and performs one transfer, after which it deasserts its request signal to notify the arbitrator that it completed its transfer. In doing so, it forfeits its remaining shares and Master 1 is granted access for three transfers, i.e., with a replenished amount of shares. Before Master 2 is granted access again, Master 1 has to continuously exhaust its shares.

This behavior shows that this arbitration mechanism is not feasible in the context of hard real-time embedded systems, as it does not guarantee access to the shared resource over a period of time, even if the master does not currently request access to it. To achieve a behavior of this kind, a modified arbitrator has been developed and introduced into the slave-side arbitration logic generated by SOPC Builder. Similar to the concepts presented in Section 4.1.3, the modified arbitration scheme uses a TDM access pattern which guarantees each master access to the slave over a predefined period of time before handing over access to the next master. This bases the decision of arbitration grants on time rather than on amounts of transfers. The properties defining the timing behavior of the developed arbitrator are the length of a slot in the TDM table and the table itself, i.e., its length in slots as well as the assignments in each slot. For example, if three masters are connected to a slave, the TDM table could have a length of ten slots, each having a length of 5 000 clock cycles. The first master could be assigned the first two slots, the second master slots three to five while the third master could be assigned the remaining five slots. In this example, Master 3 is able to access the slave 50% of the time, Master 2 30% and Master 1 has access 20% of the time. Figures 6.9a to 6.9c show examples of possible TDM tables to be used by the developed arbitration logic. As can be seen there, it is possible to leave slot assignments blank to stop the arbitrator from granting any master access during these time slots. This arbitrator guarantees each master access to the slave during their assigned time slots and generates waitrequest signals to all other masters requesting access, which makes it possible to predict the timing behavior of
Chapter 6. Development & Implementation

accesses to shared resources, thereby introducing composability into systems generated by SOPC Builder. Analog to the examples used above, Figures 6.10 and 6.11 show the timing behavior of the arbiter for two continuously requesting masters and two masters with interrupted request signals, respectively. Both show, that the granting of accesses to the slave is not decided by an amount of used shares, but by the TDM table and its slot assignments, regardless of the request patterns of the connected masters.

The following explains the modifications to the arbitration logic that achieve this behavior.

**TDM arbitrator implementation details.** In order to be synthesized for use on an FPGA, SOPC Builder generates the Avalon Switch Fabric as Very-High-Speed Integrated Circuits Hardware Description Language (VHDL) code, in which all components of the system are instantiated and, through their respective arbitrators and according to the configurations set in SOPC Builder, connected via signals. This makes it possible to not only analyze the precise arbitration scheme of each arbitrator, but to also modify its code to support the aforementioned TDM scheme. These modifications replace parts of the arbitrator code, namely the logic generating grant and chip select signals, which

67
6.1. Altera design tools

Figure 6.8: The timing of the arbitration of concurrent requests of two master ports to a single slave port. While Master 1 continuously requests access to the slave, Master 2 interrupts its requests, thereby forfeiting its arbitration shares to Master 1.

Figure 6.9: Examples of possible TDM tables of the developed arbitrator.

are used to enable the operation of the underlying slave in case a master port requests access to it. To keep the resource overhead of the arbitrator modifications at a minimal level, the TDM support is driven by two additional registers and a set of combinational operations. The registers are one 32 bit wide delay element, counting up from zero to the slot length in clock cycles and a shift register, as wide as the number of slots in the TDM table. Using the example of a memory element shared by two processors, cpu_0 and cpu_1, Listings 6.3 to 6.4 show the added parts to its SOPC Builder-generated arbitrator.

Listing 6.3: The individual grant masks and combinational logic of the TDM arbitrator

As can be seen in the signals cpu_0_grant_mask and cpu_1_grant_mask, the TDM table of this example consists of six entries, of which the first and the last three are assigned to the first processor while the second and third are assigned to the second
Figure 6.10: The timing of the modified arbitrator using a TDM arbitration scheme to grant access to the slave to two continuously requesting masters.

Figure 6.11: Analog to the example used above, two masters are requesting access to a shared slave. During the grant period of Master 2, it deasserts its request signal but keeps its grant as long as stated in the TDM table.

The grant signals are generated by reducing these grant masks and the current status of the grant shift register to a single bit using the OR operator, thereby granting access to the slave in case the single rotating 1 in the grant shift register is in the same position as any of the 1’s in the grant masks. The grant_vector signals, which is used internally to generate control signals for the address and data multiplexers as well as the waitrequest signals, is created as a concatenation of both grant bit signals. Finally, the chip_select signal, which is used to enable operation of the underlying slave, rises in case a connected master requests the slave and is granted access to it.

As shown in Listing 6.4, the grant shift register is as wide as the number of slots in the TDM table. As the TDM table in this example contains six entries, the shift register has a length of six bits and is shifted to the right every time the delay counter register reaches the value 0.
As can be seen in Listing 6.5, the delay register is set to 0 every time it reaches a predefined value, i.e., the length of a single TDM table slot in clock cycles. In this example, one slot has a length of 200 clock cycles.

Listing 6.5: The slot length counter in the TDM arbitrator

These modifications in the arbitration code for each shared slave module enable the time-based granting of accesses to these modules, thereby creating predictable access timing across the Avalon Switch Fabric. If at a point in time the arbitrator has granted access to a master, this access is guaranteed for as long as the TDM table states. The calculation of WCCTs across the Avalon Switch Fabric during these periods can therefore exclude all other masters connected to the slave module in question. In the general case, the WCCT can be determined using the exclusive access time to the slave and the longest period the master has no assigned slots in the TDM table. To calculate the WCCT for a request from cpu_0 in the example above, the length of two slots, i.e., 400 clock cycles has to be added to the slave access time. Assuming the arbitrator grants accesses to an SRAM controller block, the WCCT to read a 32 bit word are therefore 405 and 402 clock cycles, respectively. Figure 6.12 shows slave access schedule according to the TDM table.
6.1.4 System creation using Qsys

Next to SOPC Builder, the Altera design tools offer a second tool to generate Nios II-based systems, called Qsys [Altera 2013c], which is being developed as a replacement of the former. While it offers the same possibilities and user experience to the system designer, the created systems differ significantly from those generated by SOPC Builder. The biggest difference can be found in the interconnection structure that is inserted to support communication between instantiated modules. Unlike the Avalon Switch Fabric, it is not bus-based, but a Network-on-Chip which utilizes packetized blocks of data sent through Network Interfaces that act as master and slave ports. When considering the factors of predictability and composability, as is the basis of this thesis, two factors severely complicate the usage of Qsys-generated system:

- The technical documentation describing Qsys and the operation of the generated NoC do neither contain precise timing information of individual requests nor regarding interferences that are possible to occur when multiple packets need to be routed to the same destination or via the same router;
- Unlike SOPC Builder, which generates plain VHDL code possible to be modified in the way described above, i.e., to introduce composability in the arbitration mechanism of shared slave modules, Qsys does not generate code that allows for modifications in a single, shared position. Adapting one NI or packet router has repercussions throughout the complete network, which, paired with the lack of precise documentation, makes it impossible to modify Qsys-generated code to introduce composability.

Due to these reasons, SOPC Builder and its generated code are chosen as the basis of this thesis and while Altera will not offer updates for SOPC Builder in subsequent releases of its design tool suite, it will not be removed and stay a part of the supported design process.

---

1 At the time of writing this thesis, no information was found as to which version of the Altera design tools will drop support for SOPC Builder.
6.2 Creating an automated Nios II design flow

The hardware design process using the Altera tool suite presented in the previous section is based solely on graphical user interfaces while offering only limited support for automation of work flows. This, however, is a basic requirement when trying to integrate the Altera design into an automated design flow for embedded systems, as presented in Chapter 5. The following will present the introduction of support for full automation into the Altera tool suite, starting from an XML-based representation of a hardware system and the software to be implemented on it, down to the finished implementation of the system, consisting of both an FPGA programming file and fully prepared Nios II Integrated Development Environment (IDE) projects for each Nios II processor of the system.

The developed automated system creation flow, whose individual steps can be seen in Figure 6.13, has been implemented as a series of Python scripts, which make use of the existing Quartus II command line interfaces and functionality created in the context of this thesis. In addition to that, the four shaded areas in Figure 6.13 show how these individual steps are grouped together in the developed scripts. In turn, all four scripts are integrated into a surrounding fifth script and are executed with the correct parameters, which results in a fully automated flow without the need for a graphical interface or interaction from the system designer.
Chapter 6. Development & Implementation

6.2.1 Flow inputs

As shown in Figure [6.13], the input data to be given to the flow are a text-based description of the hardware system instance, which could be created by a DSE tool; and the code to be implemented on each processor, which could represent the process network mapped to its respective processing elements. To abstract from the graphical interface of SOPC Builder, an XML-based format to describe the processing elements, memory blocks and interconnection structure has been developed. An example of this format, using one processor, can be found in Listing [6.6].

```
<system>
  <!— CPUs —>
  <module kind="altera_nios2" name="cpu_0"/>
  <!— CPU-specific memories —>
  <module kind="altera_avalon_onchip_memory2"
    name="memory_cpu_0" size="32768"/>
  <!— CPU-specific UARTs —>
  <module kind="altera_avalon_jtag_uart"
    name="uart_cpu_0"/>
  <!— CPU-specific performance counters —>
  <module kind="altera_avalon_performance_counter"
    name="performance_counter"/>
  <!— CPU-specific DMA controllers —>
  <module kind="altera_avalon_dma"
    name="dma_cpu_0"/>
  <!— Communication memories —>
  <module kind="altera_avalon_onchip_memory2"
    name="comm_memory" size="4096"/>
  <!— CPU-specific connections —>
  <connection data="yes" end="memory_cpu_0" instructions="yes" start="cpu_0"/>
  <connection data="yes" end="uart_cpu_0" instructions="no" start="cpu_0"/>
  <connection data="yes" end="performance_counter"
    instructions="no" start="cpu_0"/>
  <!— DMA controller connections —>
  <connection control="yes" data="yes" end="dma_cpu_0" start="cpu_0"/>
  <!— DMA to local memory connections —>
  <connection data="yes" end="memory_cpu_0" read="yes" start="dma_cpu_0"/>
  <!— Interrupts —>
  <interrupt end="uart_cpu_0" number="0" start="cpu_0"/>
  <interrupt end="dma_cpu_0" number="1" start="cpu_0"/>
  <!— Connections to shared memories —>
  <connection data="yes" end="comm_memory" instructions="no" start="cpu_0"/>
  <connection data="yes" end="comm_memory" start="dma_cpu_0" write="yes"/>
</system>
```

Listing 6.6: An example of the text-based input to the system creation flow describing the underlying hardware structure

This format describes a system using XML syntax in the system tag and its child tags. In turn, each instance of an element in the hardware system is defined using the module tag and a set of specific attributes, which will be described below. Connections between modules, created with the connection tag, can be defined freely, i.e., each slave module can be connected to an arbitrary amount of master port, such as the on-chip memory...
6.2. Creating an automated Nios II design flow

Element `comm_memory` is connected to the instantiated Nios II/e processor called `cpu_0` and the DMA controller called `dma_cpu_0`, while the direction of the signal is defined using the `start` and `end` attributes. Similarly, interrupt signals originating from interrupt-generating modules, such as Universal Asynchronous Receivers/Transmitters (UARTs) and DMA controllers, can be defined using the `interrupt` tag.

The following will describe the hardware modules supported by this thesis to be a part of the automated creation process and how it is possible to connect them to each other.

**Supported hardware modules and connections**

The SOPC Builder supports a range of hardware modules, such as Digital Signal Processing (DSP) elements, communication controllers and peripherals. Out of this range, the following components are presently supported by the automated tool flow developed for this thesis. Care has been taken, however, to support extensibility towards both customized modules developed by the system designer as well as further modules out of the Altera design suite catalog. Section A.1 presents the method of extending this flow by adding further components.

**Nios II/e processors.** As mentioned above, out of the Nios II processor family, the Nios II/e processor shows the highest degree of predictability by avoiding features introducing timing variability. Due to this, the Nios II/e processor is the only supported core by the developed tool flow and is instantiated in the XML input format using the `module` tag and the respective `kind` attribute `altera_nios2`, as done in Listing 6.7.

```
  <module kind="altera_nios2" name="cpu_0"/>
```

*Listing 6.7: Instantiation of a Nios II/e processor core named cpu_0*

By using this, a single instance of a Nios II/e processor is created in the system, which can be connected to every slave component. Internally, these slave connections are realized by two distinct master ports: One port responsible for reading and write data to and from slave modules and one master port responsible for reading instructions from slave modules, i.e. memories holding the program to be executed. While it is only necessary to instantiate a processor core with the single `module` tag shown below, both master ports will be inserted into the system.

**On-chip memory blocks.** As explained in Section 6.1.2, Altera FPGAs are equipped with on-chip memory blocks with predictable access timing. To add a memory element of this kind to the system, a `module` tag with the `kind` attribute `altera_avalon_onchip_memory2` is used, as shown in Listing 6.8. To define the size of memory blocks in bytes, the `size`
tag is used. Additionally, the connection tag is shown, connecting the master port cpu_0 to the instantiated memory element. As can be seen there, three combinations of the two attributes data and instructions can be used when connecting a memory element to a Nios II processor. These combinations are used to distinguish between memories holding both data and instructions, only instructions or only data. The developed tool flow will utilize these attributes to internally set the connections to the two aforementioned master ports of a Nios II processor: a memory setting the instructions attribute to yes will be connected to the instruction master port of the processor while a memory setting the data attribute to yes will be connected to the data master port of the processor specified in the connection tag. As shown below, it is also possible to connect one memory to both master ports of a processor.

```
<module kind="altera_avalon_onchip_memory2" name="memory_0" size="32768"/>
1
2 <!-- Local memory to cpu_0 -->
3 <connection start="cpu_0" end="memory_0" data="yes" instructions="yes"/>
4
5 <!-- Instruction memory to cpu_0 -->
6 <connection start="cpu_0" end="memory_0" data="no" instructions="yes"/>
7
8 <!-- Data-only memory to cpu_0 -->
9 <connection start="cpu_0" end="memory_0" data="yes" instructions="no"/>
10
```

Listing 6.8: Instantiation and connection of an on-chip memory element called memory_0 with a size of 32 768 bytes and a connection to the processor cpu_0.

The TDM arbitration information, as described in Section 6.1.3, is part of the module tag of a specific hardware element, as the automated tool flow offers support for distinct TDM tables and slot lengths for each module. If this information is omitted from the module instantiation, the arbitration scheme of the automatically inserted arbitrator block is not modified. The instantiation of a shared on-chip memory element can be seen in Listing 6.9. In this example, the shared memory shared_memory with a size of 512 bytes is connected to two master ports, cpu_0 and cpu_1. To define the arbitration information, i.e., the TDM table and its slot length, the following attributes are added to the module tag: The attribute tdma_slot_length describes the slot length in clock cycles, which is set to 500 in this example, while the TDM table is defined per master port. This means, that for each connected master port, an attribute composed of the name of the master and the suffix _slots, such as cpu_0_slots is added to the tag. Its value describes the individual slots of the TDM table for this specific master, while its length denominates the length of the table itself. In this example, cpu_0 is able to access the shared memory for the first 500 clock cycles of the arbitration round, while cpu_1 is allowed access during the last 500 clock cycles. In-between, neither will receive a grant signal from the modified arbitrator. Using this method, the slave-side arbitration scheme makes it possible to define a TDM table specific to each shared slave.
6.2. Creating an automated Nios II design flow

SRAM controller. Similar to on-chip memory blocks, SRAM elements offer predictable, if slower, access timing. However, unlike on-FPGA memory elements, SRAM is a single memory block, located off-chip, and only the controller handling accesses to it is located on the FPGA itself, which makes it impossible to instantiate multiple SRAM blocks. To utilize the SRAM block, its instantiation can be seen in Listing 6.10. In this example, the external memory element is defined using the module tag and given the name sram. Its size is given as 2 MB, which corresponds to the size of the SRAM block utilized on the specific FPGA development board used for this thesis\(^2\). Additionally, the connection tag is shown, creating a connection between an instantiated SRAM controller block and the data master port of cpu_0. Similar to on-chip memory elements, it is possible to connect an SRAM block to either a data or instruction master port of a Nios II processor or to both.

JTAG UARTs. Joint Test Action Group (JTAG) UARTs offer a debugging interface to Nios II processors and are used to, i.a., download software, offer step-wise execution and facilitate communication with processor instances on an FPGA, which makes them essential tools in the development process. To add a JTAG UART instance to the system, the module tag and kind attribute altera_avalon_jtag_uart is used, as shown in Listing 6.11. In addition to that, the only required attribute is the module name, given as uart_0 in the example below. As each JTAG UART can only offer communication with a single processor, each Nios II core needs a unique UART to receive software from the host computer. As JTAG UART elements offer memory-mapped interfaces on their slave ports, the data attribute in the connection tag has to be set to yes to create a connection between the data master port of the connected Nios II processor with the slave port of the UART.

\(^2\)The FPGA used is a Cyclone IV E EP4CE115F29C7 [Altera 2013b] on an Altera DE2-115 development board [Altera 2006b], using a 2 MB SRAM chip manufactured by ISSI [Integrated Silicon Solution, Inc. 2009].
Chapter 6. Development & Implementation

Performance counters. In the context of SOPC Builder-generated systems, performance counters offer a way to cycle-accurately measure performance of individual instructions or blocks of code running on Nios II processor instances. To add a performance counter to the system, the module tag with the kind attribute `altera_avalon_performance_counter` is used, while the only required attribute is the name of the element, given as `performance_counter_0` in the example in Listing 6.12. In addition to that, it is possible to further refine the properties of the performance counter core, as shown in Altera [2007]: Using a single core, it is possible to simultaneously measure performance of up to seven individual code sections at the expense of FPGA resources. If a smaller number of concurrent sections is required, the `numberOfSections` attribute can be set. If omitted, however, the number of section is set to three.

Similar to previously mentioned slave modules, communication with performance counters is possible via memory-mapped registers offered on the slave port of the modules. Therefore, the `data` attribute of the `connection` tag has to be set to `yes` to create a connection between the module and the data master port of the respective Nios II processor.

PIO modules. Programmed Input/Output (PIO) modules offer the possibility to connect to input and output facilities on the development board, such as switches and LEDs. Internally, in the case of inputs to the FPGA, these PIO modules debounce and register signals from these resources and offer a read-only memory-mapped slave port to the Avalon Switch Fabric. In the case of outputs, these modules offer writable memory-mapped registers that can be connected to the respective output signals to drive the chosen resource. Listing 6.13 shows the instantiation of PIO modules in the system description and their connection to a master port. In the first example, an eight bit wide PIO module called `leds_green` is instantiated using the `kind` attribute `altera_avalon_pio` and, as the LEDs are driven from the FPGA the direction attribute is set to `out`. The second example, a four bit wide PIO module called `switches` is instantiated, using incoming
6.2. Creating an automated Nios II design flow

signals from switches on the development board, therefore setting the direction attribute to in.

```xml
<module kind="altera_avalon_pio" name="leds_green" direction="out" width="8"/>
<!-- Outgoing interconnection between cpu_0 <-> leds_green -->
<connection start="cpu_0" end="leds_green" data="yes"/>

<module kind="altera_avalon_pio" name="switches" direction="in" width="4"/>
<!-- Incoming interconnection between cpu_0 <-> switches -->
<connection start="cpu_0" end="switches" data="yes"/>
```

Listing 6.13: Instantiation and connection of a PIO module.

**DMA controllers.** DMA controllers offer a way for Nios II processors to delegate the transfer of large amounts of memory, thereby freeing the processor to continue execution of the program stream [Altera 2009]. To integrate a DMA controller core into the system, the core itself is split into a number of parts, which are automatically included if the instantiation of a DMA core is requested, as shown in Listing 6.14.

```xml
<module kind="altera_avalon_dma" name="dma_0"/>
<!-- DMA <-> control_slave interconnection -->
<connection start="cpu_0" end="dma_0" control="yes" data="yes"/>

<!-- DMA <-> read_master interconnection -->
<connection start="dma_0" end="memory_0" read="yes" data="yes"/>

<!-- DMA <-> write_master interconnection -->
<connection start="dma_0" end="memory_1" write="yes" data="yes"/>

<!-- optional DMA interrupts -->
<interrupt start="cpu_0" end="dma_0" number="1"/>
```

Listing 6.14: Instantiation and connection of a DMA controller core.

As can be seen there, to instantiate a DMA controller core, the module tag with the kind attribute altera_avalon_dma is used. Unlike the previous module, however, which each consisted of a single slave module, the DMA controller consists of three distinct ports to the Avalon Switch Fabric: two master ports and a slave port. While the slave port is used to receive instructions from a processor, the two master ports are necessary to facilitate reading and writing to the selected slave modules in the system. Such as a processor, these two master ports apply for arbitration grants at their respective slave port arbitrators and, when received, execute the transfer request between those memories. To create the necessary connection to the three individual ports, the connection tag of a DMA controller core introduces three distinct attributes: control, read and write. To
connect a Nios II processor to the control slave of a DMA controller, the control attribute has to be set to yes. To connect a memory to the read master port of a DMA controller, the read attribute has to be set to yes and finally, to facilitate writes originating from the DMA controller, its write master port is connected to a memory if the write attribute is set to yes. Additionally, the controller is able to generate interrupt signals to connected Nios II processors every time a memory transfer has been completed. This interrupt connection is specified using the interrupt tag and, similar to the connection tag, uses start and end attributes to define the source and target for the interrupt signal as well as the number tag, specifying the interrupt number on the Nios II processor.

**FIFO memories.** First In, First Out (FIFO) memories are a memory element of the Avalon design tools, facilitating FIFO-like storing of data by one or multiple masters. Similar to a DMA controller, a FIFO memory block, instantiated as seen in Listing 6.15, consists of multiple ports to the Avalon Switch Fabric: one slave port to read data from the FIFO and a second slave port to write data to it. If required, a third slave port can be instantiated offering a control interface and providing the fill status of the FIFO, which can be used to test for the availability of data in case of a reading access or a full FIFO in case of a write. This port, however, can be omitted, as the FIFOs are configured to use a feature called backpressure: To avoid the overwriting of contents, a full FIFO raises a waitrequest signals to the writing master, similar to the same signal of arbitrators in the system in case a master cannot be granted access. If this signals is raised during a write transaction, the master is blocked until the signal is cleared, thereby making the overwriting of values by writing into a FIFO impossible. The same is valid for reading accesses to an empty FIFO memory. In addition to the name of FIFO block, the instantiation can be refined using a depth attribute, setting the depth of the FIFO. If this attribute is omitted, the value is set to eight.

```xml
<module kind="altera_avalon_fifo" name="fifo_0" depth="16"/>
<l!-- Writing FIFO access -->
<connection start="cpu_0" end="fifo_0" data="yes" direction="in"/>
<l!-- Reading FIFO access -->
<connection start="cpu_1" end="fifo_0" data="yes" direction="out"/>
<l!-- Optional FIFO control slave -->
<connection start="cpu_1" end="fifo_0" data="yes" control="yes"/>
```

Listing 6.15: Instantiation and connection of a FIFO memory element.
Processor code

Next to the text-based system description presented above, the second input to the developed architecture creation flow is code for each processor. As will be explained in Section 6.2.5, the code contains the functions to be executed on these processors and, in addition to that, an [HAL] which includes drivers for the slave modules connected to the respective processor as well as code necessary to start the processor, interrupt handling routines and emulate unimplemented hardware instructions, such as multiplication and division operations. The code itself can be written in either C or C++, while support for the latter can be omitted in order to save resources.

The following will give a detailed description of each step of the creation flow, how it makes use of the input data and finally, how a complete hardware architecture is created.

6.2.2 SOPC Builder model creation

The first and biggest step in the architecture creation flow is the translation of the text-based system description into a format usable by SOPC Builder. As mentioned earlier, SOPC Builder does not support the automation of the hardware creation process but only offers a graphical user interface. While this gives the designer the possibility to easily create multiprocessor systems, it is unusable in the context of automated system creation, such as, e.g., after the DSE phase of an automated embedded systems design flow, as presented in Chapter 5. SOPC Builder does, however, use an XML-based format to describe all details of a created system, which includes every component and its instantiation details as well as every connection between them.

SOPC Builder file format

The basic system description format presented above is, in its separation of modules and connections, inspired by the format in which SOPC Builder saves its systems. However, it only contains the essential and most basic description of the system. The notable differences between the two formats are listed in the following.

Complete parameter list. Each module contains every parameter to the underlying entry in the SOPC Builder module catalog. For example, while the instantiation of a memory element in the basic description only contains its name and size, the complete description contains 20 individual parameters that either have to be derived from surrounding modules or are standard values for the specific FPGA.

Memory base addresses. Each connection does not only contain its start and end point in the system, but also the base address for memory-mapped accesses to it. Using the
example of a memory connected to a Nios II processor, the connection tag describing
the connection of the slave port of the memory module with either the data or instruction
master of the processor contains the baseAddress attribute, describing the address in
the memory of the master module to which the slave is mapped. This value depends on
all other slaves connected to this master and on their size in memory, i.e., the size of
the memory or the number of memory-mapped registers they offer to the master port.
In addition to that, the base address of each slave module has to be the same in every
master it is connected to.

Address maps in master ports. Similar to the base address of slave modules in their
connection tags, each master port contains an attribute describing the precise address
map of all slave modules connected to it. This mapping includes the base address of each
slave module connected to the master as well as the range of addresses it spans in its
memory. Every Nios II processor and DMA controller contains two slave maps, one for
each master port they utilize.

Clocking information. A feature omitted from the basic system description is the
possibility to define multiple clock domains in a single SOPC Builder-generated system.
However, at least one clock source needs to be present and connected to each module in
the system.

Device-specific information. In addition to the system itself, the SOPC Builder format
contains information specific to the FPGA on which the system is to be implemented.
This information includes the FPGA family and type as well as information about its
capabilities.

Translation algorithm

The pseudo code of the developed script to translate the basic system description into
an SOPC Builder-compatible format and to generate the information not present in the
basic format can be examined in Algorithm 6.1. In addition to creating the required
output format, the algorithm creates a graphical representation of the system in the
GraphViz format [Gansner, Koutsofios, and North 2010], including all instantiated Nios II
processors, DMA controllers, and slave modules. The listed algorithm only gives a short
representation of the developed code, however, offering the basic ideas behind the
translation.
6.2. Creating an automated Nios II design flow

**Input:** Basic system description

**Result:** [SOPC] Builder-compatible system description and graphical representation

```python
def create_sopc_model(basic_system_description):
    set basic system properties;
    add clock source;
    # adding modules and clock connections
    for module in input system:
        if known module found:
            add module to output system using get_element(current_module);
            read and set correct parameters;
            add clock connection;
    # create two-dimensional representation of system called slave_list
    # to drop need to parse input file
    for connection in input system:
        if master already in slave_list:
            add slave and its address range to master list;
        else:
            add master to slave_list;
    # update parameters of processor in the output file
    for master in output file:
        set parameters;
        write slave maps using create_slave_map(slave_list[current_master]);
    # write all connections to output file
    for master in slave_list:
        for slave in slave_list[master]:
            add connection from master to slave
```

**Algorithm 6.1:** The translation of the basic system description into a [SOPC] Builder-compatible format

To add the required code for individual modules to the output file, the script uses a library of [XML] descriptions called `xml_element_sources`, which contains the [SOPC] Builder modules listed above, named according to the `kind` attribute of the `module` tag. To add the modules to the output file, the developed function `get_element` is called using the `kind` attribute and a dictionary of parameters to be changed in the [XML] element, such as the size of a memory to be added or the depth of a [FIFO] block. Using this, the steps performed in the process are:

1. Basic information, such as the device family and type as well as the name of the Quartus II project, is written into the output file;
2. A clock source for the system is added to the output file;
3. The input file is parsed for each `module` tag and `get_element` is called using the `kind` attribute and the dictionary of parameters to be changed, writing the extended descriptions into the output file;

82
4. The input file is parsed for each module a second time while connections between the clock source and the modules are added;
5. An intermediate representation of the system as a two-dimensional list is created. The first dimension is formed by the master ports of the system while the slaves connected to each of them form the second dimension. Additionally, each slave contains its memory size span and base address. This representation eliminates the need to parse the input file repeatedly;
6. The parameters of each processor and DMA controller are adapted according to the set of slaves connected to them, as described above. A developed function create_slave_map is called for each master port, creating the address map of connected slaves and writing it into the master module. In this step, a unique requirement of SOPC Builder has to be satisfied, which states that the base address of module has to be aligned to its memory span, e.g., if a memory element has a size of 2 048 bytes, its base has to be divisible by 2 048. A representation of this function can be found in Algorithm 6.2;
7. After writing all required parameters into the master modules, the connections between the modules are added to the output file. This step, as explained above, uses the base addresses of each slave module calculated in the previous step and adds them to the connection tags in the output file.

```
def create_slave_address_map(master, slave_list):
    base_address = 0;
    for slave in slave_list[current_master]:
        # Handle SOPC Builder requirement that the base addresses of modules have to be aligned to their address span
        if (base_address % slave_span) != 0:
            base_address += slave_span - (base_address % slave_span);
        add base_address and span into output string;
        base_address += slave_span;
```

Algorithm 6.2: The creation of the slave address map for a master port

With completion of the execution of the algorithm, encapsulated in a developed Python script called create_sopc_model.py, a system representation in a SOPC Builder-compatible file format has been created, which is used to create the complete system. During this process, the generated system description will be verified and can, if necessary, be adapted using the graphical interface, which will automatically open in case a verification error occurred. After successful completion of this verification step, VHDL

---

3It has to be noted here that even the automatic setting of base addresses and slave maps by SOPC Builder itself is not always able to produce valid results. In this case, the designer has to manually alter the base addresses in the created system.
sources for each module as well as the Avalon Switch Fabric interconnection structure are created. These sources are then used in the next step of the creation process, in which the arbitration mechanism of the generated arbitrators is modified.

6.2.3 Arbitrator modification

To introduce the notion of composability to the Avalon Switch Fabric, the arbitrators generated by SOPC Builder need to be modified to support a TDM access arbitration scheme, as explained in Section 6.1.3. The challenges connected with the development of the script performing this step of the creation flow are the analysis of the generated arbitrators and the TDM details encoded in the basic system description as well as the creation of valid VHDL code to complete their modification. To accomplish this, the script works in the following way:

1. The basic system description is analyzed and slaves connected to multiple master ports are identified;
2. For each of these modules, the TDM table, i.e., the slot length and individual slot assignments, are extracted and checked for validity. If, for example, multiple masters are assigned to the same slot, the script will notify the designer and stop the execution of the flow;
3. The VHDL source code of the interconnection structure is then analyzed and, if the generated arbitrator module of a previously identified slave is found, the following modifications will be performed:

   (a) A number of signals have to be added to the code: grant_delay, a vector with a length of 32 bit, grant_shift_register, as wide as the number of connected master ports to the slave as well as a grant_mask and granted signal for each connected master;

   (b) The grant_mask signals for each slave are filled with the verified information provided in the basic system description, e.g., cpu_7_slots="-X--" is translated into cpu_7_grant_mask <= "0100";

   (c) For each master port, the grant_mask signal and grant_shift_register are combined using the AND function. The result of this operation, effectively indicating if this master currently holds the grant signal of the arbitrator, is then reduced using OR and assigned to the granted signal of this master;

   (d) The processes describing the behavior of the grant_shift_delay counter with the appropriate slot length and the grant_shift_register shifter are inserted. The latter implements the shift operation using code that is independent of the length of the shift register;
(e) The grant_vector and chip_select signals of the arbitrator are adapted. The former is a concatenation of the granted signal of each master port while the latter is a combination of request and granted signals of each master. Listing 6.16 shows the creation of the chip_select signal in the developed script;

(f) As the last step, the original definitions of the grant_vector and chip_select signals are removed from the arbitration code.

```python
# the chip_select has to be built using all connected masters
# it is only high when a master requests the slave AND is granted access to it
# an OR relation connects these AND pairs for each master
print '\t' + current_slave_module + '_chipselect <=
counter = 0
for connected_master in master_list[current_slave_module]:
    print '(internal_' + connected_master + '_requests_' + current_slave_module + ' AND internal_' + connected_master + '_granted_' + current_slave_module + ')
    if counter != (len(master_list[current_slave_module]) - 1):
        print 'OR ' if counter == (len(master_list[current_slave_module]) - 1):
            print ';
        counter = counter + 1
Listing 6.16: The creation of the chip_select signal for an arbitrary number of master ports. Using the Python fileinput module, each print statement is printed directly into the output VHDL file

The developed script, called modify_arbiters.py, implementing these steps makes it possible to modify the arbitration code of slave modules connected to an arbitrary number of masters, an essential requirement in the creation of a hardware architecture template. After the successful completion of these steps for each arbitrator, the modified VHDL code is synthesizable and can be inserted into a by Quartus II project, which will be created in the following step of the creation flow.

6.2.4 Quartus II project creation

Unlike SOPC Builder, the Quartus II design suite offers a Tool Command Language (Tcl)-based interface to automate common tasks required in the design process of hardware design. The following lists the automation steps used in the context of this thesis.

Project creation. To create an empty project and assign basic properties, such as setting the FPGA family and specific device, the scripting interface quartus_sh with the --prepare option and the project name is used, which creates an empty project and enables further processing.
6.2. Creating an automated Nios II design flow

Setting timing constraints. During the synthesis of a hardware system, Quartus II performs static timing analysis to identify critical paths throughout the system and check for possible timing violations. To optimize this process, it is possible to specify the VHDL signal carrying the clock and its frequency, which is accomplished by using the quartus_sta static timing interface and a developed Tcl script that executes the required commands to add the clock to the system and set its frequency.

Setting location assignments. In order to facilitate communication between the FPGA and development board, it is necessary to map the VHDL signals to pins of the FPGA package. This step is accomplished by a Tcl script executed via the quartus_sh scripting interface. Additionally, a constraint is set, which instructs Quartus II to place all generated design files in the output folder in the project. While this is not a required step, it simplifies the layout of the project.

Generating top-level VHDL code. Next to the automatically generated system code, the created project needs to contain a top-level VHDL source file, instantiating the SOPC Builder-generated system and describing the interface of the FPGA to the development board. As both the name of the project and the generated system can be set freely by the designer and are therefore subject to change, a top-level source file using these names has to be created. This is accomplished by a Python script, called prepare_vhd1.py, which modifies a standard top-level source file using the correct names and places it in the project folder.

6.2.5 Nios II project creation

To enable the development of software for Nios II-based systems, a customized version of the Eclipse IDE [Altera 2011b] is used. For each processor, two Eclipse projects are created: one project containing the actual code to be executed on the processor and another project encapsulating the HAL particular to this processor, called Board Support Package (BSP), which contains all device drivers and support code to execute software on the processor. Due to this, the contents of the BSP depend on information about the specific processor configuration and all slave modules connected to it, which is extracted from the SOPC Builder system description. To automate these tasks, the Nios II IDE offers command line interfaces for each required step in the creation of these two project types. These tools have been encapsulated in a developed Python script called create_nios_projects.py, which performs the following steps:

1. In order to create projects for each processor in the system, the basic system description is parsed and each processor is identified using its kind attribute. Using the name of the processor, a BSP is created using the nios2-bsp command line...
tool, which can be customized to the needs of the application. In the context of this thesis, the BSPs are configured to create smaller device drivers and communication libraries, which reduces the code footprint of the required system software on each processor [Altera 2011b, p. 171];

2. After a BSP has been created for each processor in the system, a second project for code development is generated using the nios2-app-generate-makefile tool. This project does not contain any code but consists of only a generated makefile, which, when called, directs the compilation of the BSP project and every source code file in a given list of directories forming the software to be executed on the FPGA. The standard location for source files is a subfolder in the creation flow directory structure. In addition to that, the makefile for each project makes it possible to download the compiled software to its respective Nios II processor on the FPGA using the download-elf target.

After both a BSP and development project have been created for each processor, the script prints a number of messages to assist the system designer with initiating connections to the instantiated UARTs, start compilation of created code or to simulate the design using ModelSim [Altera 2012a].

6.2.6 Combined architecture creation flow

All steps described shown in Figure 6.13 and described above are encapsulated in a surrounding Python script automating the execution of each step and providing each script with the correct parameters according to the settings chosen by the designer. These settings give the possibility to choose a name for the project to be created, which can be set using the required --project_name option, from which all internal names will be derived; the option to force SOPC Builder to open its graphical user interface after the system has been created to allow the designer to review and, if needed, modify the system, chosen with --sopc; and the --no_tdma option, which can be used to disable the modification of the automatically generated arbitration logic for testing purposes.

If all individual steps in the creation flow have been executed successfully, the FPGA will be programmed with the created image file and the designer can begin to download developed code to the instantiated Nios II processors using the generated makefiles. Figure 6.14 shows the directory structure of the project creation flow, including all developed scripts.

6.2.7 Conclusion

Using the developed creation flow for Quartus II projects enables to automated generation of predictable and composable Nios II/e-based hardware systems and the required supporting software structure. This makes it possible to integrate the platform and its
6.3 Integration into the existing ForSyDe design flow

In order to integrate a developed predictable hardware system into the automated embedded systems design flow presented in Chapter 5, a number of factors have to be considered: The automated generation of hardware systems from text-based descriptions has to be supported. As discussed in the previous section, this thesis presents such a flow for a modified SOPC Builder-generated hardware architecture. Further, the platform creation, which takes place after completion of the DSE phase, has to be connected to the output of this phase and generate a system supporting the execution of ForSyDe process networks both in hardware and in software. In the context of this thesis, the focus was lain on providing support for the execution of SDF process networks, as their analysis provides a large amount of information directly relating to the hardware architecture that aims to implement them, as explained in Section 5.1.3.
6.3.1 DSE phase results

As discussed in the previous chapter, the DSE phase forms the most computationally extensive part of an automated embedded systems design flow as it needs to consider not only the mapping of processes including all resulting timing, resource usage and power considerations, but also to create a schedule for both communication and computation of tasks on individual processing elements [Lahiri, Raghunathan, and Dey 2004; So, Hall, and Diniz 2002]. The DSE tool for the ForSyDe design flow currently in development aims to generate both mapping and scheduling information for ForSyDe SDF process networks and therefore creates a set of results needed to accomplish the creation of a hardware architecture to execute the process network under analysis. As, at the time of writing this thesis, an easily analyzable output format of this step did not exist, the format shown in Listing 6.17 has been developed. For the simple SDF process network seen in Figure 6.15 the shown output can be generated.

![Figure 6.15: A simple SDF process network to be analyzed by the DSE tool of the ForSyDe design flow.](image)

As can be seen there, the highest hierarchical level is formed by processes using the process tag. In turn, each process contains a number of SDF channels, i.e., communication paths to other processes. In the example, both processes, process 0 and 1, communicate using one channel each. These channels contain the size of a single token to be sent, the buffer size on the receiving side as well as the number of the receiving process. In the example, one channel sends tokens with a size of 40 bytes to process 1 and requires four buffer elements to hold the received information. The second channel sends tokens with a size of 60 bytes from process 1 to process 0, also requiring four buffer slots to hold the received information. The mapping of SDF processes to processing elements can be seen in the cpu attribute of each process. In the example, process 0 is mapped to processor 0, while process 1 is mapped to processor 1, therefore requiring two processing elements on the hardware system to be generated from this result set. Additionally, the memory requirement each process, including its stack and, if analyzable heap data, is encoded in the local_memory_requirement tag.
6.3. Integration into the existing ForSyDe design flow

Listing 6.17: An example of the XML-based output format of the DSE phase of the ForSyDe design flow

This format contains all required information to create a hardware platform instance on which execution of the analyzed process network is possible.

6.3.2 Proposed architecture layout

While the platform template presented in the previous section supports a wide range of possible structures of processors and slave modules, it is necessary to define a fixed architectural layout to be created in the context of the ForSyDe design flow to be able to extract cost and timing characteristics to be taken into account during the DSE phase. Figure 6.16 shows the chosen configuration that has been created from the example used above. In this configuration, each instantiated processing element is equipped with both a JTAG UART and performance counter as well as a DMA controller to perform the transfers of input and output token to external memories. To enable fast and predictable access to all local data and instructions, each Nios II/e processor is equipped with a private on-chip memory block, which eliminates both the need for arbitration when accessing local data and, as all data is stored on the fastest memory elements, for caches on the instruction or data master port of the processor. To facilitate communication with other processors in the system, one shared on-chip memory block is instantiated for each SDF channel. Due to this, each slave module will never be shared by more than two master ports, thereby
Figure 6.16: An example architecture created from a DSE result set using two processor elements and one performance counter, JTAG UART and DMA controller each. To implement communication across two SDF channels, two shared memory blocks are instantiated equipped with two TDM-based slave-side arbitration modules.

decreasing the WCCT across channels using these modules, as the longest time a master is forced to wait for its arbitration grant is only inferred by one other master competing for access to the slave. Each shared memory is instantiated with the exact size to hold the amount of tokens stated in the DSE result set.

6.3.3 Software support

In ForSyDe, using the concepts of MoCs computation and communication are treated separately by the analysis and implementation steps of process networks. While the code describing the functionality of processes in ForSyDe SystemC can be easily extracted from the network and inserted into code to be executed on the developed architecture, the communication infrastructure is profoundly different. This necessitates the creation of architecture-specific software support to enable communication between ForSyDe processes executed on the same processing element and across shared memories. In its SystemC-based implementation, ForSyDe uses FIFO buffers to connect the outputs of processes with the respective inputs that receive their data. The same approach has been taken in the context of this thesis: Each instantiated shared memory acts, with a supporting software layer, as a FIFO memory for tokens with arbitrary size while the size of the memory block itself will be, as mentioned above, chosen to be able to hold the correct number of tokens and the management information necessary to support the FIFO operation. For tokens of a very small size, 4 bytes, the communication infrastructure supports the usage of hardware FIFO blocks which decrease the WCCT for channels.
utilizing them as no software support is necessary to manage the fill level of the FIFO.

To send a token from one process to another, a function called `send_token()` has been implemented, which internally handles all implementation details such as calculation of memory base addresses, data copying of token data and management of modules involved in the transaction, such as hardware FIFO buffers or software-managed FIFOs on on-chip memory blocks. Similarly, to receive a token, the function `receive_token()` has been implemented.

**Implementation details**

In order to send and receive tokens, the developed functions require three parameters:

- `unsigned int sender_id`: The first argument given to the communication functions is the number of the sending process, which has to be set as a part of the design flow to be unique throughout the system. To send data from process 0 to process 1, the first argument therefore has to be set to 0;
- `unsigned int receiver_id`: The second argument required by the communication functions is the unique number of the receiving process. In the example given above, the second argument therefore has to be set to 1;
- `void* data_address`: The third parameter is used differently by the sending and receiving functions. In the case of the former, a pointer to the token to be sent while in the case of the latter, a pointer to a position in memory to store the received token needs to be provided. Using the `void` pointer type makes it possible to send and receive arbitrary data types and structures, as long as their address and length in memory is known.

Internally, both functions block the execution of the program flow when trying to write to a full buffer or read from an empty buffer, respectively. This effectively enables support for the execution of self-timed schedules, as explained in Section 5.4.

**Common data.** To be able to send and receive data via shared memories connected to pairs of processors, the send and receive functions need to decode the base addresses of these memories from the parameters provided in the function calls. This is made possible by a set of matrices containing all information required to perform reading and writing of tokens via shared memories. These hold the size of a single token, the number of tokens in the particular buffer, as well as the base address of the memory or, if used for this particular channel, of the hardware FIFO. This information is stored in a header file to each processing element. An excerpt of the header file of the processing elements of the example used above can be seen in Listing 6.19 while Listing 6.18 shows the data structure used to store FIFO management information on the shared memories.
Chapter 6. Development & Implementation

Listing 6.18: The CircularBuffer data structure used to maintain the FIFO management information

```c
typedef struct {
    int size;   // maximum number of elements
    int start;  // index of oldest element
    int count;  // index at which to write new element
} CircularBuffer;
```

This data structure holds the size of each software FIFO buffer as well as information relating to the current fill status of the buffer. One instance is positioned at the beginning of each shared memory in the created system by an initialization function using the buffer_length information found in Listing 6.19.

Listing 6.19: The CPU-specific information required to address shared memories and initialize software FIFO data structures

```c
#define NR_PROCESSES 2

volatile CircularBuffer* address_map[NR_PROCESSES][NR_PROCESSES] =
{
    {0, (volatile CircularBuffer*) SHARED_MEMORY_PROCESS_0_PROCESS_1_BASE},
    {((volatile CircularBuffer*) SHARED_MEMORY_PROCESS_0_PROCESS_1_BASE, 0)}
};

volatile int* fifo_list[NR_PROCESSES][NR_PROCESSES] =
{
    {0, 0},
    {0, 0}
};

unsigned short token_sizes[NR_PROCESSES][NR_PROCESSES] =
{
    {0, 40},
    {60, 0}
};

unsigned short buffer_length[NR_PROCESSES][NR_PROCESSES] =
{
    {0, 4},
    {4, 0}
};
```

Sending function. The sending function performs a number of steps when it is called to send data from one process to another, which are explained in the following:

1. Check the entry fifo_list[sender][receiver] of the fifo_list data structure whether the current channel uses a hardware FIFO to transmit the data. If this is the case, a write command to this FIFO structure is issued and the processor blocks due to the aforementioned backpressure feature of the hardware FIFO modules until a slot is available;
6.3. Integration into the existing ForSyDe design flow

2. If the transmission is based on a software FIFO in the shared memory, the function reads the `address_map[sender][receiver]` and `token_sizes[sender][receiver]` information to prepare the transmission of the local token to the shared memory;

3. If the FIFO is currently full, i.e., if the `start` and `count` entries in the data structure have the same value, a wait command is issued which blocks execution of the program flow until another processor is able to extract the stored information and empty a slot in the buffer;

4. If at least one slot is available, the function updates the fill status of the FIFO, calculates the target address inside the shared memory and issues a transmit request to the DMA controller of the processor which executes the copying process.

Receiving function. The receiving function works in a very similar fashion as the sending function. First, the `fifo_list` matrix is checked for a hardware FIFO implementing the channel and, if existing, a read command is issued to it. If a shared memory and software-managed FIFO is used, the above listed steps are taken in the same order and data is copied from the correct position in the shared memory to the location provided by the `data_address` parameter in the function call.

6.3.4 Creating a system from a DSE tool result set

In order to integrate the developed architecture in the ForSyDe design flow, an automated translation of the DSE phase output format presented in Listing 6.17 into a basic system description and accompanying processor-specific header files has been implemented. To accomplish this, it operates in the following way:

1. Parse the input file and create an intermediate representation of all SDF channels and their respective data as well as of all required processing elements, of which each is one out of three categories:
   - the processor contains only processes communicating over channels with a token size of 4 bytes, therefore neither shared memories nor a DMA controller need to be instantiated for this processing element;
   - the opposite is the case, i.e., the processor contains only channels using tokens larger than 4 bytes, therefore no hardware FIFO blocks but a DMA controller and shared memories need to be instantiated;
   - the processor contains processes using both kinds of channels, therefore hardware FIFO blocks as well as DMA controllers and shared memories need to be instantiated;

2. Using the intermediate representation and processor information, instantiate all required modules in the output file, i.e., in the basic system description;
3. In the same manner, create all connections between instantiated modules while special care is taken in the following cases:

- If a DMA controller is present for the particular processor, connect its read and write master ports to the processor and respective shared memories as well as its control slave port to the processor;
- If hardware FIFO blocks are used, connect its input and output slave ports to the processor instances to which the processes utilizing the particular channel are mapped.

4. After the system description has been created, the intermediate representation is used to create customized C header files and C source files containing the structure for each instantiated processing element, enabling the usage of the `send_token` and `receive_token` functions. In this process, the following steps are taken:

   (a) According to the three categories listed above and due to the fact that each BSP only contains software support for modules which are connected to the respective Nios II/e instance, the header file for each processing element includes either drivers for the hardware FIFO modules and the DMA controller or only for the latter. Additionally, to enable a generic implementation of the send and receive functions, preprocessor directives and conditional compilation are utilized to differentiate between the cases of driver availability;

   (b) Similar is valid for the base addresses of shared memories which make use of macro definitions in the BSP to each processor, as only base address macros of shared memories connected to the processor in question are able to be added to the header file;

   (c) In the last step, the matrices describing token sizes and buffer lengths are added to the file, while, in case of a connected DMA controller, it has to be guaranteed that token sizes are divisible by four.

After the successful completion of the generation process, the created system description and header files are then able to be used as both the input for the automated system creation flow as well as for the preparation of ForSyDe SystemC process code to be executed on the architecture instance.

6.4 Conclusion

This chapter presented an analysis of the Altera design suite and its system creation tool SOPC Builder in terms of timing and predictability as well as the modifications necessary to introduce the notion of composability to systems created using this tool suite.
Further, to enable the integration of SOPC Builder-generated systems into an automatic design flow for embedded systems, the steps necessary to achieve complete automation of the system creation from a text-based description to an FPGA programming file and projects enabling software design are presented as well as their implementation using a set of Python scripts. In addition to that, the connection to a proposed the Design Space Exploration phase of the ForSyDe system design flow is presented, which enables the automatic generation of system descriptions and processor-specific header information from its results.

In the following chapter, the validation and performance test efforts using two example applications conducted in the context of this thesis will be presented.
Validation & Case Studies

The following section will present test results based on two example applications, implemented as ForSyDe-SystemC process networks and executed on the developed hardware architecture. Additionally, the integration of the architecture template with the generic Tahmuras framework and the SWEET Worst-Case Execution Time analysis tool will be discussed.

7.1 Test structure and goals

To be able to validate the behavior of the developed architecture in both timing and functionality, multiple tests will be conducted on a number of varying platform instances. The test applications will be a parallel JPEG encoder which has been developed as a ForSyDe process network in the context of this thesis and an image analysis algorithm called SUSAN, first presented in Smith and Brady [1997] and adapted as a ForSyDe process network in Altinel [2013]. In addition to that, two external tools which are able to provide complementary functionality within the ForSyDe system design flow will be presented: Tahmuras, a generic DSE tool, and SWEET, a WCET analysis solution. For both tools, support for the architecture template presented in this thesis has been developed.

In the context of these test efforts, the starting point is an executable functional specification of the algorithm, developed as a ForSyDe process network. As discussed in Chapter 5, the functionality of each process is wrapped in pragma directives and will be copied to the platform code. Send and receive instruction around these extracted blocks will be manually added to enable communication between processor instances. Using Tahmuras and the obtained WCET values of each process in the ForSyDe process network by SWEET, a set of DSE results will be obtained and required information converted into the model description shown in Section 6.3.1. The presented automatic conversion into a system description and the subsequent creation of an FPGA image and Nios II projects as well as the programming of the FPGA will enable the acquisition of both performance test data as well as data enabling conclusion about predictability and composability of the platform. The former will be realized using the presented performance counter slaves
modules for each processing element. The latter will be performed by comparison of individual process network runs on the platform with runs of the same process network and platform instance while memory-intensive applications accessing all shared memories will simulate unpredictable elements running on the same platform. To able to justifiably claim composability, delays in the execution of the process network must not occur.

7.1.1 **SWEET** WCET analysis tool

_Swedish Execution Time Tool (SWEET) [Engblom, Ermedahl, and Stappert 2000; Engblom, Ermedahl, and Stappert 2001]_ is a low-level WCET analysis tool which, for a given hardware architecture, is able to predict both best-case and worst-case execution time bounds on given program flows. The analysis of these programs, developed in C, is based on an intermediate representation of their control and data flow, called _Artist Flow Analysis (ALF) language [Gustafsson, Ermedahl, and Lisper 2011]_. This representation, independent of both programming language and hardware implementing the program, is analyzed by SWEET for its loop bounds and infeasible paths, i.e., paths in the control flow which cannot be reached by any set of input data due to calculations and assignments during execution of the program [Gong and Yao 2010]. After evaluating these paths and their length to identify the longest and shortest among them, i.e., the best-case and worst-case program timing [Altenbernd et al. 2011], SWEET makes it possible to relate these paths to costs of the individual ALF language instructions.

**Extracted ALF construct costs.** In order to be able to generate BCET and WCET values for the developed architecture template, ALF construct costs have to be extracted for Nios II/e processor instances. These constructs directly relate to common CPU instructions, such as _Arithmetical-Logical Unit (ALU) commands, branches and load and store operations_. Table [7.1] gives a list of these construct as well as their cost in clock cycles on Nios II/e instances. As explained in Section 6.1.1, these processors do not exhibit interdependencies in the timing of instructions due to the lack of branch prediction, pipelined instruction execution and multiplication logic.

**Annotating algorithm input data.** SWEET gives the designer the possibility to set _abstract annotations_ in the beginning of the analysis process, i.e., initial values for global variables. As explained in Section 2.2.2, simulation-based WCET testing solutions heavily rely on a set of combinations of the input data to the tested program and measure its execution time for each combination. While SWEET does not belong to this category but is an abstract flow-control analysis tool able to produce safe execution time bounds, its gives the possibility to refine the analysis of the program stream by setting _abstract_ initial values for global variables. These values can either be specific values or ranges for variables, i.e., setting an initial global counter to 0 or measured input data to always lie
### Table 7.1: The ALF language construct costs on a Nios II/e processor instance.

<table>
<thead>
<tr>
<th>ALF construct</th>
<th>Cost in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>All ALU operations</td>
<td>6</td>
</tr>
<tr>
<td>Multiplication and division</td>
<td>1 502</td>
</tr>
<tr>
<td>Modulo</td>
<td>1 502</td>
</tr>
<tr>
<td>Shift</td>
<td>38</td>
</tr>
<tr>
<td>All logic operations</td>
<td>6</td>
</tr>
<tr>
<td>All branches</td>
<td>6</td>
</tr>
<tr>
<td>Load and store operation</td>
<td>6</td>
</tr>
<tr>
<td>Function calls</td>
<td>5</td>
</tr>
</tbody>
</table>

between 0 and 255; or can be a higher-level description, i.e., setting a variable to possibly contain all integer values. Listing 7.1 shows the annotation of a C data structure using both abstract and specific values.

```c
typedef struct {
    int content[8];
    int block_id;
} example_structure;

element_structure example;

// annotation of initial values to be used by SWEET
STMT_ENTRY main::entry 0 ASSIGN example 0 32 8 TOP_INT
|| example 256 32 1 INT 0 256;
```

**Listing 7.1:** Annotating SWEET input data.

In this example, the data structure `example_structure` contains an array of eight integer values and an identification number, also stored as an integer. The instance of this data structure called, `example` is declared. The next and last line in Listing 7.1 show the annotation of this data structure instance to be used by SWEET. When entering the main function of the code to be analyzed, which is not shown in this example, the values for `example` are set as follows: the range of bit 0 to bit 32 is set to the abstract value TOP_INT, thereby possibly containing all integer values for analysis, which is repeated eight times. In the next steps, the range of bit 256 to 288 (256 + 32) is set to possibly contain all integer values between 0 and 256. For the testing purposes of this thesis, this process has been performed for each process in the ForSyDe process network of the developed sample applications.

In order to facilitate the testing of multiple functions, a makefile has been developed, executing all required steps in the determination of execution time bounds.
## 7.1. Test structure and goals

<table>
<thead>
<tr>
<th>Template element</th>
<th>Cost in LEs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nios II/e processor</td>
<td>3 600</td>
</tr>
<tr>
<td>TDM bus</td>
<td>100 · ports</td>
</tr>
<tr>
<td>Hardware FIFO buffer</td>
<td>25 · depth</td>
</tr>
</tbody>
</table>

### (a) Area costs

<table>
<thead>
<tr>
<th>Transmission using WCCC for $n$ bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDM bus</td>
</tr>
<tr>
<td>Hardware FIFO</td>
</tr>
</tbody>
</table>

### (b) Timing costs

| $1200 + 48n + \max(TDM)$            |
| $(12 \cdot \frac{n}{4})$            |

Table 7.2: The extracted information describing the timing behavior of the developed architecture template with regard to the Tahmuras DSE framework.

### 7.1.2 Tahmuras DSE tool

Tahmuras [Attarzadeh Niaki, Mikulcak, and Sander 2013] is a generic DSE framework, enabling the characterization of predictable platform templates using a set of abstract characterizations. These component and platform templates make it possible to, i.a., define service functions and cost indicators for individual components of a platform template and of the platform as a whole. Additionally, a process network to be implemented on the platform template can be captured, forming the basis for implementation decision. Tahmuras then makes it possible to convert these abstract characterizations into a constraint satisfaction problem, which in turn can be solved using a constraint problem solving tool [C. Schulte, Tack, and Lagerkvist 2013] in regard to any cost indicator expressed by the abstract template. Using this, a DSE problem can be solved to, for example, minimize the area necessary to implement the system or to minimize the length of a process network round by instantiating a larger number of processing elements.

In the context of this thesis, the developed hardware platform template for the ForSyDe system design flow has been captured as an abstract platform template to be used by Tahmuras. The individual components describe processing elements, the TDM interconnection structure and hardware FIFO modules. Table 7.2 shows the extracted costs for both the instantiation of modules as well as the timing behavior of transmission across the TDM bus. As can be seen there, each cost indicator is not fixed but coupled to a parameter which depends on the surrounding system. For example, the FPGA area cost of the TDM interconnection structure is decided by the number of connected master ports while the cost of a FIFO hardware buffer is calculated using its depth. The same is valid for the timing behavior of both transmission over the TDM bus as well as FIFO buffers.

**Processing element costs.** A single processing element in the context of Tahmuras consists of all parts listed in Section 6.3.2: A Nios II/e processor (1 050 LE), a DMA controller (1 600 LE), a performance counter (670 LE), a UART (160 LE) as well as a private memory whose control logic consists of 170 LE.
TDM structure costs. As explained above, each slave module connected to the modified Avalon Switch Fabric necessitates the instantiation of an arbitrator, which grows in size, the more master ports are connected to it. In the context of the architecture structure proposed by this thesis, this results in an overall cost for the TDM interconnection structure of 100 LE per connected master port and slave module.

Data transmission costs. The timing impact of the transmission of data over both the TDM interconnection structure as well as hardware FIFO modules has been determined both statically using SWEET as well as through measurements. Using SWEET the software implementation of FIFO buffers via shared memories has been analyzed, which, as explained in the previous chapter, consists of the update of FIFO management values, which in turn uses multiplication and addition operations on the target platform. Using both methods, the Worst-Case Communication Cycles (WCCC) of \( n \) bytes over the modified Avalon Switch Fabric has been determined as

\[
(1200 + 48n + \max(TDM)) \text{ clock cycles},
\]

in which the latter describes the maximum time the maximum time the master port aiming to read or write does not receive the arbitration grant from the TDM arbitrator. If, however, a single transaction takes longer than the duration of all assigned time slots in a TDM round, this formula becomes invalid, as the master is interrupted while transferring and has to wait for arbitration again. One possibility to mitigate this is to make guarantee that the DSE phase assigns time slots and slot lengths large enough to hold a single transfer, or to extend the formula using a modulo operation with the allotted slot time. This, however, adds to the complexity of the DSE analysis and should therefore be avoided in favor of the former solution.

On the other hand, as each hardware FIFO in the developed architecture template represents a single channel and therefore a single direction of transfer, read and write transfers across these channels do not need to be arbitrated. Thus, its timing behavior is not dependent on the arbitration details and is translated into a single read or write operation to a memory-mapped register. These operations consist of an address calculation and the subsequent read or write operation, both performed in six clock cycles each, therefore resulting in a WCCC to a hardware FIFO module of twelve clock cycles per word.

7.2 Sample applications

The following section will present the sample applications, test results and conclusions used to validate functionality and timing behavior of the developed architecture template and creation flow.
7.2. Sample applications

7.2.1 Parallel JPEG Encoder

The developed sample application is a parallel Joint Photographic Experts Group (JPEG) image encoder, implemented as process network in Forsyte SystemC using the SDF MoC. Its input is a bitmap raster graphic, which is divided into blocks of 16×16 pixel, called Minimum Coded Units (MCUs). These blocks are fed into a variable number of processing pipelines, which form the first level of parallelism in the implementation of the encoder application. Each processing pipeline consists of three steps performing the encoding and compression functionality of the algorithm, of which the second step is split into six simultaneous units, forming the second level of parallelism. This structure with a block-level parallelism of four can be seen in Figure B.2. The following will give an overview of the individual processes in the network and their functionality.

**Reading input file blocks.** This process, called bitmap_reader reads blocks of 16×16 pixel of the input raster image and outputs them to the processing pipelines. Additionally, it sends the number of the current block through a delay element back to itself, which it uses to track the current state of the input file reading. Furthermore, this block identification is used to initialize the reading process via an initial value on the SDF delay element.

**Block collection and unzip.** An implemented block called rgb_block_collector collects blocks of input data from the bitmap reading process, creates a tuple of blocks, which in turn is sent to an unzip process, where it is split and sent to the processing pipelines. The number of elements in the tuple as well as output signals of the unzip process are determined by the level of block parallelism chosen by the designer, as shown in Figure B.2.

**Color conversion and subsampling.** The first step in each processing pipeline is the color conversion of each block, called color_conversion, of input data to the Y’CbCr format, representing the chroma and luminance components of each pixel of the input file. This makes it possible to split and simultaneously perform the subsequent steps of the encoding process on each component. Two components are formed by the Cb and Cr parts of the input block, while four are formed by 8×8 pixel blocks of Y, the luminance component of the input block. Thus, the output signal of the color conversion process is a six-tuple of these components, which is split into its elements by an unzip and sent to the subsequent steps of the encoding process.

**Discrete Cosine Transformation.** The second step of the processing pipeline is formed by the Discrete Cosine Transformation (DCT) of each of the six color components, called dct. Therefore, this process is instantiated six times in each processing pipeline.
The dct process transforms each color component into the frequency domain which makes it possible to filter elements that cannot be identified by the human eye, therefore forming the first basis for compression of a raster image into a JPEG file.

**Quantization and entropy coding.** The same is valid for the third step of the processing pipeline, the quantization and entropy coding of each color component, called huffman. This step performs the aforementioned filtering of high-frequency changes in the input image by analyzing the output of the previous DCT step and setting elements over a certain limit to zero. This makes it possible to perform the subsequent steps of the encoding process, the lossless compression of created bit streams by removing repeated zeros and replacing them with an end of block command.

**Merging encoded color components.** The final step of the processing pipeline is the merging of the bit streams representing each encoded color component. This process, called concatenate_steps, merges these bit streams, saved internally as arrays of integer values, and sends them to writing process.

**Writing the output file.** The final step of the overall encoding application is the writing of merged bit streams to the output JPEG file, called jpeg_write. Using the block identification number of the input block, which is relayed through the processing pipeline, this sorts the encoded blocks and writes them to the output file. Additionally, a set of reserved identification numbers are used to signal the initialization of the writing process, i.e., the writing of JPEG header information to the output file, as well as the closing of the file, thereby finished the operation of the process network.

**Worst-case and best-case execution time bounds**

Using SWEET, each function in the developed ForSyDe process network has been analyzed with regards to its worst-case and best-case execution cycle bounds, shown as Worst-Case Execution Cycles (WCEC) and Best-Case Execution Cycles (BCEC). Table 7.3 shows these bounds as determined by SWEET utilizing the extracted ALF construct costs for the developed platform. As explained above, the analysis of each functions requires the setting of abstract initial values, which has been done using information about the context of each function, i.e., limiting the block identification number as well as image dimension inputs to the jpeg_write process according to possible sizes of input images. As can be seen from these results, while the execution time of a number of functions is highly dependent on the input data, three functions do not suffer from this timing variability. In these cases, the best-case equals the worst-case execution time bound. Additionally, to decrease the effect of the timing variability introduced by the software emulation of multiplication, division and modulo operations, the ALF construct costs for these
7.2. Sample applications

<table>
<thead>
<tr>
<th>Process</th>
<th>BCEC</th>
<th>WCEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_bitmap</td>
<td>328 833</td>
<td>628 199</td>
</tr>
<tr>
<td>rgb_block_collector</td>
<td>15 273</td>
<td>15 273</td>
</tr>
<tr>
<td>color_conversion</td>
<td>833 549</td>
<td>1 778 125</td>
</tr>
<tr>
<td>dct</td>
<td>234 907</td>
<td>375 675</td>
</tr>
<tr>
<td>huffman</td>
<td>243 765</td>
<td>389 728</td>
</tr>
<tr>
<td>concatenate_steps</td>
<td>220 335</td>
<td>8 508 471</td>
</tr>
<tr>
<td>jpeg_write</td>
<td>82 341</td>
<td>142 912</td>
</tr>
</tbody>
</table>

Table 7.3: The worst-case and best-case execution cycle bounds on the developed JPEG encoder application as determined by SWEET.

operations, seen in Table 7.1, have been adapted according to the data types used by the individual functions. For example, multiplications in the color_conversion function are only performed on data types with a width of eight bit, which makes it possible to set the WCET of a single multiplication operation to a value of 376. Similarly, the width of shift operations is limited to three, the execution time of a shift operation can thus be set to nine.

Test case 1

The first basis for testing the implementation uses a version of the JPEG encoder with a single processing pipeline and therefore without block parallelism. In this instance, the process network to be implemented on the architecture template consists of 16 processes, which can be seen in Figure 7.1. Due to the size of the processing network and the resulting complexity of the DSE analysis problem, it was not possible to obtain a result using Tahmuras. Therefore a mapping of the process network to four processing elements has been chosen manually, thereby making use of the parallelism inside a processing pipeline by mapping three pairs of dct and huffman steps each on two processing elements and utilizing the remaining processors to read the input image and write the output file. To implement this mapping solution, ForsyDe SystemC zip and unzip processes to multiple receiving processes are omitted, as they can be replaced by multiple send_token and receive_token function calls on the target platform, without the need to combine or split multiple signals.

The hardware implementation and C header files have been automatically generated from a created DSE output file describing all 16 processes and 12 distinct channels, as presented in Section 6.3.1. Each channel has been translated into a shared memory, which are accessible via the appropriate send_token and receive_token function calls. The software implementation uses the automatically generated header files, including all required drivers for DMA controllers and performance counters. Additionally, all relevant
Chapter 7. Validation & Case Studies

Figure 7.1: The first test case of the JPEG encoder application omitting block-level parallelism but implementing the simultaneous processing through pairs of dct and huffman blocks. RB marks the bitmap reader process, CC the color conversion, DCT and H the DCT and Huffman encoding processes while CS and WJ mark the step concatenation and JPEG writer processes.

Figure 7.2: The test pictures used by the JPEG encoder application examples.

functionality, as indicated by the ForSyDe SystemC pragma directives, has been imported into the CPU-specific code for the Nios II processor instances on the target platform. The input images, seen in Figure 7.2, have a size of 120,054 bytes and a dimension of a 200×200 pixel, therefore consisting of 156 distinct MCUs to be converted by the JPEG encoder. As the implementation of the process network uses a fixed quality setting, the size of both output image lies at 20 kB, which is the same in every implementation.

Resource requirements. The created architecture instance uses four processing elements and 12 shared memories, which predicts a worst-case FPGA resource utilization of 16,400 LE while the implemented system uses 15,697 LE. Each processing element is assigned a private memory with a size of 40 kB, with the exception of the first processor, which uses a 200 kB memory to store the input image. Additionally, all twelve SDF channels utilize a combined amount of 4,352 bytes of memory. This results in an overall on-FPGA memory utilization of 329.25 kB.

Execution time measurements. In order to test the validity of the statically determined execution time bounds, each process on the target platform has been wrapped into performance test logic to extract measurement of its execution time. Table 7.4 shows the
results of these measurements and their deviation from the values statically determined by SWEET. As can be seen there, the determined bounds are kept throughout the processing of all blocks of the input images, however, the deviation of the measured results from the determined bounds shows inconsistent quality. While the best-case results and bounds of the dct and huffman processes show close proximity to each other, the worst-case bounds of all but the jpeg_write process are greatly exaggerated by SWEET. The reason for this lies in the dependence of the length of multiplication and shift operations on their operands, as both operations are used extensively throughout all processes of the JPEG encoder.

**Communication time measurements.** In this test, each shared memory arbitrator uses the same TDM details: A table consisting of slots, each 10,000 clock cycles long. During the first time slot, the sending participant of the channel is allowed to read and update the management information stored in the shared memory; during the second time slot, the DMA controller of the same processor is allowed to write the token into the shared memory while during the third slot, the receiving processor is allowed to read the token and update the management information. Therefore, \( \max(\text{TDM}) \) for all three participants is 20,000.

To be able to measure the exact time needed to transfer a token using the implemented send_token and receive_token functions, the DMA controllers are disabled by setting the DMA preprocessor macro to 0, thereby forcing the utilization of memcpy and therefore the processor itself to send and receive tokens. Table 7.5 shows a comparison of the measured and predicted Worst-Case Communication Cycles for a range of token sizes, including 48 and 144, which are the token sizes sent between the processors of the example mapping in this test case. Each measured value shown is the largest out of the set of transfers occurring during the processing of the input images. As can be seen there, the predicted bounds on the transmission times are kept throughout the complete
Table 7.5: Worst-case communication cycles as predicted by the formula given above and measured on the target architecture.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>WCCC during grant</th>
<th>WCCC without grant</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Predicted</td>
<td>Measured</td>
</tr>
<tr>
<td>4</td>
<td>1 392</td>
<td>1 183</td>
</tr>
<tr>
<td>8</td>
<td>1 584</td>
<td>1 365</td>
</tr>
<tr>
<td>32</td>
<td>2 736</td>
<td>2 497</td>
</tr>
<tr>
<td>48</td>
<td>3 504</td>
<td>3 207</td>
</tr>
<tr>
<td>144</td>
<td>8 112</td>
<td>7 507</td>
</tr>
</tbody>
</table>

compression of the input images. In addition to that, two test cases are presented for each token size: In the first case, the communication takes place exclusively while the master is granted access to the shared memory, which makes it possible to ignore the arbitration details in the prediction, i.e., max(TDM). The second case presents the more general solution of a master trying to communicate just after its grant signal expired. In this case, the maximum impact of max(TDM) can be observed in both the prediction and measured values. However, no violation of the prediction takes place.

The same is valid for the execution time predicted and measured for hardware FIFO-based transfers. While there is no SDF channel with a sufficiently small token size to support the instantiation of a hardware FIFO, a dummy channel carrying token with a size of four bytes has been introduced and its performance evaluated. The results confirm the prediction made above: since a transmission to a hardware FIFO using the implemented send_token and receive_token functions results in a single address calculation and register read or write, the execution always lies at 12 clock cycles per word.

While Table 7.5 shows the validity of WCCC predictions over the modified Avalon Switch Fabric depending on the size of the token to be transmitted, Table 7.6 compares the communication time of a single token with the fixed size of 144 bytes when transmitting it with and without TDM modifications. To accomplish this, a second master is configured to continuously read and write to the shared memory used for transmission by the first master. Additionally, its slots in the standard round-robin arbitration scheme are varied to increase the shares the first master receives to access the memory. As can be seen there, series of tests using the TDM arbitration scheme show a consistent WCCC for both masters whereas the unmodified arbitration scheme makes it possible for the second master to delay the transmission process, which is prohibited in a composable system. The first column shows both masters having a single share in the round-robin arbitration, resulting in a large delay, which decreases the more shares the first master receives.

While this first test case and mapping aimed to validate predictions about the execution time of individual tasks as well as worst-case communication times across the
7.2. Sample applications

Table 7.6: The measured communication time via an unmodified Avalon Switch Fabric compared with the arbitrator design developed in this thesis.

<table>
<thead>
<tr>
<th>Shares</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fairness-based</td>
<td>8 341</td>
<td>8 144</td>
<td>7 962</td>
<td>7 769</td>
<td>7 558</td>
</tr>
<tr>
<td>[TDM]-based</td>
<td>7 507</td>
<td>7 507</td>
<td>7 507</td>
<td>7 507</td>
<td>7 507</td>
</tr>
</tbody>
</table>

Table 7.7: The predicted as well as measured results. The measured application round time shown is formed by the average over all 156 [MCU] compression rounds.

<table>
<thead>
<tr>
<th>Result</th>
<th>Predicted</th>
<th>Measured</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource usage (LE)</td>
<td>16 400</td>
<td>16 233</td>
</tr>
<tr>
<td>Application round time (µs)</td>
<td>237 454</td>
<td>219 443</td>
</tr>
</tbody>
</table>

[TDM]-based interconnection structure, the second test case using the developed JPEG encoder uses mapping and scheduling solutions created using Tahmuras and aims to assess their validity.

Test case 2

To decrease the complexity of the DSE problem, the second basis for testing combines each processing pipeline instance to a single process. This makes it possible to utilize Tahmuras to identify mapping solutions for the chosen JPEG encoder configuration, which can be seen in Figure 7.3. In this instance, block-level parallelism is employed by utilizing four block processors. In addition to a valid mapping including predictions about resource requirements of the target architecture, Tahmuras creates schedules for both computation and communication, using WCEG and WCCC values input with the SDF process network layout, resulting in a predicted worst-case process network round time. To obtain a solution, Tahmuras has been configured to minimize the resource utilization on the target architecture while keeping a throughput requirement, which results in a solution utilizing four processing elements. If the throughput requirement was omitted, the proposed solution would consist of only a single processor. Table 7.7 compares the predictions made by Tahmuras to their measured counterparts on the implemented architecture. As can be seen there, both predictions lie close to the measurements and are therefore validated by the target architecture and implementation of the process network.

7.2.2 SUSAN image analysis

The second sample application is Smallest Univalue Segment Assimilating Nucleus (SUSAN), an image analysis tool set first presented in Smith and Brady [1997] and implemented as a ForSyDe process network in Altinel [2013]. Its utilization in the context of the testing
efforts of this thesis aims at porting a second and foreign ForSyDe SystemC process network to the developed architecture. The set of algorithms encompassed in SUSAN is able to filter noise in images and detect edges and corners, out of which the edge detection algorithm is used in the context of this thesis. The ForSyDe process network can be seen in Figure B.1. The following presents the three distinct functions in the process network implementing the functionality of the edge detection algorithm.

Reading input file blocks. Similar to the JPEG encoder application presented above, the first step is the splitting of the input raster image file into blocks to enable iterative processing of the input data, which is performed by a function called getImage.

USAN calculation. The next step is the calculation of the Unvalue Segment Assimilating Nucleus (USAN) for each pixel of the image block using a function called Usan. This structure describes a set of pixels sharing the following properties: They lie in an area surrounding the current pixel of interest and have similar brightness as this pixel. Additionally, the brightness values of all matching pixels are combined into a single USAN value, which, when compared with a picture-wide threshold set by the algorithm, makes the detection of edges in the image possible.

Edge direction calculation. In order to be able to more accurately detect the edge, this step detects its direction by calculating the stochastic moment of all USAN values in the current block using a function called Direction.

Thinning. Using the directions of edges calculated in the previous step, this function, called Thin, thins and refines every edge in the block by eliminating all non-maximum values around edges and estimating sub-pixel information.
7.2. Sample applications

<table>
<thead>
<tr>
<th>Process</th>
<th>Determined BCEC</th>
<th>Determined WCEC</th>
<th>Measured BCEC</th>
<th>Measured WCEC</th>
<th>Deviation in % BCEC</th>
<th>Deviation in % WCEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>getImage</td>
<td>936 272</td>
<td>1 038 000</td>
<td>1 120 671</td>
<td>511 334</td>
<td>19.70</td>
<td>50.74</td>
</tr>
<tr>
<td>Usan</td>
<td>511 452</td>
<td>18 325 980</td>
<td>606 452</td>
<td>3 152 780</td>
<td>18.57</td>
<td>82.80</td>
</tr>
<tr>
<td>Direction</td>
<td>1 246 039</td>
<td>96 622 687</td>
<td>1 458 581</td>
<td>49 491 337</td>
<td>17.06</td>
<td>48.78</td>
</tr>
<tr>
<td>Thin</td>
<td>160 170</td>
<td>1 335 010</td>
<td>192 728</td>
<td>255 968</td>
<td>20.33</td>
<td>80.83</td>
</tr>
<tr>
<td>putImage</td>
<td>14 082</td>
<td>84 592</td>
<td>16 792</td>
<td>49 526</td>
<td>19.24</td>
<td>41.45</td>
</tr>
</tbody>
</table>

Table 7.8: Worst-case and best-case execution cycles of the SUSAN edge detection algorithm as determined by SWEET and measured on the target platform. Additionally shown is the deviation of all the measured values from the statically determined results.

Figure 7.4: The chosen mapping solution for the SUSAN algorithm using two processing elements and shared communication memories. In the platform implementation, multiple input signals to one process originating from a single process are encapsulated in C data structures.

Writing the output file. In the final step, using a function called putImage, the edge information of each block is drawn into the output raster image file.

Worst-case and best-case execution time bounds

SWEET was utilized to determine worst-case and best-case cycle bounds on the execution of each process in the SUSAN ForSyDe-SystemC implementation. These results can be seen in Table 7.8.

Mapping solution

After capturing the process network in a Tahmuras-compatible format, a mapping and scheduling solution has been generated. However, due to the linear operation of the edge detection algorithm without parallelization, the only proposed solution is that of a single processing element. Although it provides the highest performance for the given process network, the mapping solution chosen is that of two processing elements and can be seen in Figure 7.4. While this configuration does not provide the highest possible performance, it aims at demonstrating the implemented communication facilities across the TDM interconnection structure and their predictable timing.
Chapter 7. Validation & Case Studies

(a) Input image.  (b) Detected edges

Figure 7.5: The test picture used by SUSAN and the created result image.

Results

The SUSAN process network has been ported to the target architecture configured to utilize the mapping solution described above. The input image and the end result, i.e., an image containing all detected edges, can be seen in Figure 7.5 while a comparison of the best-case and worst-case execution cycles for each process and their deviation from the statically determined results is shown in Table 7.8. As can be seen there, the best-case execution cycle estimations show a consistent overestimation of approximately 20% while the worst-case bounds show a strong deviation of up to 83%. As explained above, the reason for this can be found in the software emulation and therefore factor length dependence of the multiplication operations. However, all determined execution cycles bounds are kept.

7.3 Conclusion

The most important properties claimed by the developed architecture, composability and predictability, have been examined in three distinct test setups created using two applications: a developed JPEG encoder with support for parallelity as well as SUSAN, an edge detection algorithm. The composability of the platform could be confirmed using a comparison of transmissions over the modified and unmodified Avalon Switch Fabric. The predictability of the platform in both timing and resource usage was evaluated and confirmed for the example applications. However, as the utilization of the SWEET timing analysis tool showed, it is difficult to measure a degree of predictability due to dependencies of the platform timing on the application.
This chapter summarizes the work performed in the context of this thesis as well as its results and achievements. Its second part delivers an outlook on possible future work to continue the topics introduced by this thesis.

8.1 Summary

Within this thesis, a predictable and composable hardware architecture template based on the Altera design tools and Nios II/e processing elements has been presented. Further, its integration into the ForSyDe embedded systems design flow has been discussed and an implementation was realized and used to generate test results on the basis of two distinct applications. In the following, the steps taken to achieve these goals are listed.

Background study. In order to be able to analyze and evaluate predictability and composability of the existing Altera MPSoC creation flow, a study into the theoretical background of predictable hardware architectures has been conducted. Additionally, two current predictable hardware and software architectures taking profoundly different approaches were presented: CompSOC, a NoC-based architecture using MicroBlaze soft cores and Merasa, a bus-based architecture, using custom components throughout the whole system.

Evaluation of Altera design tools. Using the conclusions drawn in the background study, the Altera design tools, especially the Nios II-based MPSoC design flow has been analyzed towards its support for predictability and composability. This step showed the necessity of modifications in the automatically created interconnection structure, the Avalon Switch Fabric, and possible sources of WCET overestimation in the implementation of arithmetical operations.

Developing an automated creation flow for Nios II-based system. To be able to integrate systems based on the Nios II soft processor into an automated embedded
systems design flow, it is necessary to automate their creation from a text-based input format. As the Altera design tools do not offer support for this, an overlying flow has been developed, encompassing all required steps in the creation of Nios II-based system, i.e., the creation of a hardware system, its synthesis into an FPGA image file and programming of the FPGA as well as the creation of software development projects including all required drivers for each instantiated processor.

**Integration into the ForSyDe design flow.** One possible design flow to house the developed platform, the ForSyDe system creation flow, was introduced and discussed with regards to its compatibility to the developed platform. Using this as a working basis, the integration of the architecture template and its automated creation flow into the ForSyDe design flow has been presented. This includes the software support for the execution of process networks via communication functions and the conversion of a higher-level DSE phase result set into a platform instance description and processor-specific information on which the software support is based.

**Validation.** The final steps of this thesis work were taken by evaluating the claims of predictability and composability made by the developed architecture. In order to accomplish this, two ForSyDe-SystemC applications were tested: a developed parallel JPEG encoder and an edge detection algorithm. The basis for comparison of the test results generated using these applications was formed by SWEET, a WCET analysis tool and Tahmuras, a DSE tool able to capture generic predictable architectures and process networks. The collected results confirmed both the composability as well as the predictability of the developed architecture template and software support for the execution of SDF process networks. One reservation, however, is connected to the latter due to the high amount of overestimation of the WCET bounds generated by SWEET which stems from the software-based emulation of multiplication instructions.

### 8.2 Future work

This thesis and its architecture template forms the first step in the integration of a predictable MPSoC in an automated ForSyDe embedded systems design flow. The remaining steps and a discussion of possible future work to be performed following this thesis are outlined in the following.

**Creating a customized hardware module to accelerate arithmetical operations.** As mentioned above and in the previous chapter, the software emulation of a number of commonly-used arithmetical instructions introduces a danger of gross overestimation of the execution time of processes by timing analysis tools. One possibility to mitigate this is
the creation of hardware support for these operations, called a custom instruction in the context of the Altera design tools. This idea encompasses a specialized hardware module for multiplication and division of a set of numbers, connected to Nios II processors and able to be addressed via instruction in the C code. While this enables the substitution of visible multiplications with customized and therefore faster instructions, the acceleration of hidden operations of this kind would not be possible, such as in the calculation of bus target addresses.

Replacing the processor. While Chapter 6 showed that the Nios II/e processor core introduces no timing unpredictability due to its lack of extended features such as pipelining, caching or branch prediction, its performance when executing arithmetical instructions is low when compared to the larger members of the Nios II processor family. As the discussion in Chapter 6 showed, high predictability cannot be achieved using the Nios II/f core while the Nios II/s processor introduces considerably less timing variations. Due to the lack of precise documentation about its branch prediction, pipeline timing and caching, it was dismissed in the context of this thesis. However, these extended features can be the target of further analysis, especially when considering that it is possible to omit the caching capabilities of Nios II/s processors when using tightly-coupled memories. Using these and a carefully created set of tests, the extraction of precise timing information and therefore the possibility to create an abstract model describing the processor type can be possible.

Automatic ForSyDe-SystemC process code extraction. An important step in the complete automation of the ForSyDe design flow is the automated extraction of process code implementing the functionality of each element in the process network and the insertion of communication code. While the creation of required header files to support the operation of the mentioned send and receive functions is performed automatically, the insertion of code in the processor code is not. As ForSyDe-SystemC supports the extraction of C code from its processes, the remaining step lies in the generation of local data structures on each processor hosting the input and output data of processes and the insertion of communication function calls with the appropriate sender and receiver information.

Extending software support for further MoCs. The proposed architecture layout and software support focuses on the execution of SDF and HSDF process networks. ForSyDe and its SystemC modeling framework, however, encompass a wider range of Models of Computation, such as the Synchronous MoC mentioned in Chapter 5. To extend the support to this MoC, the communication functions need to be extended to produce absent value tokens in case a buffer to be read is currently empty.
8.3 Conclusion

The work performed in the context of this thesis shows that it is possible to create a predictable hardware architecture based on the Altera design tools and automate their creation in order to integrate them into an automated system design flow. Further, their integration into the ForSyDe SDF process network flow lays the basis of its extension to additional MoCs as well as to further automate the design process.
Bibliography

Articles, theses & books


— (June 2013). The ForSyDe Homepage. URL: https://forsyde.ict.kth.se/.


Technical documentation


— (May 2013a). *Avalon Interface Specifications*.


Appendices
A.1 Extending module support of the system creation flow

In order to extend the range of supported modules, the following steps have to be taken:

1. Using SOPC Builder, a system has to be manually generated using the required module, in this example, the *Altera Avalon Mutex core*.

2. The SOPC Builder-generated system description, here called `system.sopc`, contains a *module* tag describing the instantiation of the mutex core including all parameters to be set by the system designer. From this file, the following information is to be extracted:
   - The content of the *module* tag has to be saved in an XML file to be used by the automatic system creation flow. In this example, the *kind* attribute of the mutex core is called `altera_avalon_mutex`, so a file called `altera_avalon_mutex.xml` is to be created;
   - The name of the clock input of the module in question has to be extracted by examining the *clock* connection tag connecting the mutex core to the system clock. In this example, the clock input port is called `clk`;
   - Similarly, the name of the data and address input of the slave module has to be extracted using the *connection* tag between the module and the Avalon Switch Fabric, here called `s1`;
   - Finally, the memory address span of the module needs to determined by examining a slave map of a processor the module is connected to. In this example, the line `<name="mutex_cpu_0.s1" start="0x200000" end="0x200008">` of the first processor contains the required information.

3. Using these four items, the `create_sopc_model.py` python script can be adapted to be able to generate a mutex core in the following way:
   - The created XML file containing the module instantiation needs to be placed in the `lib/xml_element_sources` folder;
A.2 Creating a system from the DSE output

(b) The code containing the input file parsing has to be adapted to correctly identify the mutex core by its kind attribute in the basic system description and set its parameters, such as the initial value of the mutex;

(c) The same adaption is necessary for the second loop, the adding of clock connections to the output file, using the name of the clock input of the mutex core slave module;

(d) The final step required is the adaption of the Avalon connection creation in the script by extending it to be able to identify the mutex core by the end attribute in the connection tag of the basic system description and to create the data and address channel using the correct input name.

These required steps need to be taken in order to extend the support to further slave modules in a system. The integration of further master modules, such as additional types of DMA controllers or the Altera RapidIO communication system, however, requires more delicate changes to the creation flow and cannot be summarized in short form.

A.2 Creating a system from the DSE output

The following lists the steps required to create a complete system including FPGA image file and code projects from a DSE phase result set using the example of the very simple SDF process network shown in Figure 5.3:

1. The first step is the creation of a DSE output file to be processed by the dse_to_architecture.py python script. The contents of this file can be seen in Listing A.1. As in the process network, two processes and a single channel connecting them are defined, carrying tokens with a size of 40 byte;

2. To generate CPU header files and a system description from this file, the dse_to_architecture.py script needs to be called while pointing to the file using the --input_file command line argument;

3. The script generates a set of folders, one for each processor in the system, containing two files: buffers.h, containing all required drivers and memory locations of the shared memories for the particular processor, and cpu_n.c, which includes the header file and offers the basic structure of computation and communication to be executed on the processor. In addition to that, the script creates the XML description of the system to be created;

4. To create the system to implement the process network, the created XML file needs to be copied into the create_quartus_projects folder structure, i.e., into the lib folder, and the script can be executed taking the --project_name and --output_location command line arguments.
5. After the script successfully finished its execution, the connected FPGA is programmed and the designer can copy the created processor-specific folders into the input_files directory of the project creation flow. These locations have been added to the makefiles of each code projects, which reside in the software subfolder of the created project.

```
<dse_output>
  <process id="0" cpu="0">
    <channel>
      <to_process>1</to_process>
      <token_size>40</token_size>
      <buffer_size>4</buffer_size>
    </channel>
    <local_memory_consumption>32768</local_memory_consumption>
  </process>
  <process id="1" cpu="1">
    <local_memory_consumption>32768</local_memory_consumption>
  </process>
</dse_output>
```

**Listing A.1:** The DSE result set describing the simple process network consisting of two processes and a single data channel.

### A.3 Code availability and documentation

Every developed python script, C code and ForSyDe process network can be found in the Subversion repository of this thesis. Additionally, documentation is generated using Doxygen-compatible comment blocks and can be created from the provided doxygen configuration file.

#### A.3.1 License

All in the context of this thesis developed and described code is licensed under the BSD 3-Clause License, a copy of which can found in [http://opensource.org/licenses/BSD-3-Clause](http://opensource.org/licenses/BSD-3-Clause). This document itself is licensed under the Creative Commons Attribution-NoDerivs 3.0 Unported (cc by-nd 3.0) License, which can be read in [http://creativecommons.org/licenses/by-nd/3.0/](http://creativecommons.org/licenses/by-nd/3.0/)
A.3. Code availability and documentation
Figure B.1: The process network implementing the SUSAN edge detection algorithm
Figure B.2: JPEG encoder process network