UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

Validation Method for Array Processor in Space

Validierungsmethode für einen Array Prozessor in der Raumfahrt

Masterarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Sven Ott

ausgegeben und betreut von
Prof. Dr. Martin Leucker

Die Arbeit ist im Rahmen einer Tätigkeit bei der Firma Airbus DS GmbH entstanden.

Lübeck, den 15. April 2016

Abstract

The High Performance Data Processor (HPDP) is a novel processor architecture for stream-based data processing applications in space systems. It combines a coarse-grained reconfigurable dataflow array, sequential processing units, and space-oriented peripherals. Flexibility and an in-orbit programmable core provide a full range of processing capabilities for a long operational lifetime.
This work analyses the current development state of the HPDP chip, followed by establishing a suitable on-board test method for commissioning. In particular, the test concept covers critical internal components, memories, as well as interfaces at the processor's operational speed using the instruction set. The correctness of the developed test programs is demonstrated in ModelSim using an accurate design of the HPDP.

Zusammenfassung

Der High Performance Data Processor (HPDP) basiert auf einer neuen Prozessortechnologie für Stream-basierte Datenverarbeitungsprogramme in der Raumfahrt. Es vereint ein konfigurierbares Datenfluss-Array, sequenzielle CPUs und raumfahrttaugliche Schnittstellen. Eine breite Anwendung und lange Einsatzzeit sind durch die flexible Programmierung während einer Mission gewährleistet.
Die vorliegende Arbeit analysiert den aktuellen Entwicklungsstatus des HPDPs und stellt darauf aufbauend eine geeignete Testmethode für die Inbetriebnahme vor. Das verwendete Konzept beinhaltet das Testen von kritischen internen Komponenten, Speichermodulen und Chip-Schnittstellen unter der operativen Geschwindigkeit des Prozessors. Die entwickelten Testprogramme wurden in ModelSim mit einem vollständigen HPDP Design simuliert.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

_____

Lübeck, 15. April 2016

Acknowledgements

I would like to give special thanks to Tim Helfers for giving me the opportunity to write this thesis at Airbus Defence and Space and the guidance throughout the period.

Thanks to Prof. Dr. Martin Leucker from the Institute for Software Engineering and Programming at the University of Lübeck for supporting my work at the other end of Germany.

Big thanks to Daniel Bretz and Volker Baumgarte for their great help to understand the HPDP architecture quickly and work more efficient with the different simulations tools.

Also, great thanks to my friends Jonathan Felix and Kevin Böckler for their advice to make this thesis even better.

Contents

1 Introduction

The demand for high-performance earth observation satellites is significantly growing. New sensor technologies with increasing resolutions and rising instrumental data rates require a cost-effective, real-time performance of on-board data management and high-speed downlink to earth. A high-performance data processor is required, which can handle the rising amount of data, and further, can lower costs of other on-board equipment by reducing downlink data. Moreover, new transmission formats and standards demand in-space reconfigurability to avoid a limitation of satellite's functionality for its whole life. [SHW+08]

A suitable hardware architecture must support pipelining and data flow parallelism on the one hand, and on the other, offer an in-orbit programming flexibility for the processing components. Application-Specific Integrated Circuits (ASIC) is an advanced architecture providing high operating performance while energy consumption is low. Flexibility might be reached by implementing multiple functionalities on a single or multiple connected chips. However, the lack of reconfiguration capabilities is an inevitable drawback of ASICs. A more flexible architecture are Field Programmable Gate Arrays (FPGA) based on SRAM, offering reconfiguration at the gate level, but they need more space and a higher power consumption at less operational frequency than an ASIC. Also, there are some concerns about radiation effects, and internal data can not be stored during reconfiguration. Another approach solving the arising problems might be a digital signal processor (DSP). Its architecture is specialised for data-processing applications, and the algorithms are coded in software. While a high flexibility and in-orbit reconfiguration is given, high power consumption is disadvantageous. Moreover, the computational performance for stream-based applications is not as high as FPGAs or ASICs. [BEM+03, SHW+08]

A new emerging technology is an array processor that combines the benefits of other established architectures. A high computational performance with low power consumption, while being reprogrammable at any time, makes that processor class a promising technology for the earth observation field. The eXtream Processing Platform (XPP) architecture is based on such a reprogrammable array processor technology and forms the core of the High Performance Data Processor (HPDP), that is used throughout this work. The flexible and reliable architecture can process a high amount of data concurrently and is re-

programmable at runtime. Pipeline, instruction level, data flow, and task level parallelism fulfil the requirements of heterogeneous applications in fields such as telecommunication, simulation, digital signal processing, and cryptography [PAC09g, SAH13].
Moreover, a new radiation-hard, 65-nm semiconductor technology is used for manufacturing the HPDP chip. This technology is not space-proven yet.

The HPDP comprises various complex and heterogeneous components those are arranged on a single chip. A whole set of testing challenges has arisen through the long development process. Signal integrity (SI) must be preserved in digital circuits as such problems can cause a failure to operate at the planned operational speed, that makes the circuit unreliable. Known causes are crosstalks, ringing, ground bounce or noise in the power supply. Those may lead to wrong or delayed signals, or reduce the lifetime of processor chip due to additional hardware stress.

1.1 Purpose of the Thesis

The goal of this work is to develop a test strategy bringing the new HPDP chip with its new semiconductor technology into operation. A commissioning test plan shall be established along with the on-board test software to verify the reliability of internal features and interfaces of the new array processor technology. Meanwhile, the first HPDP demonstrator chip is in the manufacturing phase and, therefore, not provided throughout this thesis. Suitable verifications are required to ensure the reliability of developed commissioning tests. Simple usability and good extensibility are essential because later the test method will be used by an HPDP chip reviewer.
The processor utilises a new manufacturing process of a 65-nm radiation-hard technology which also must show its reliability in practice. However, testing of the physical persistence under extreme conditions in space is not part of this thesis. Whereas, stress tests under normal circumstances shall be included.

1.2 Structure of the Thesis

The first section introduces the HPDP architecture from its top level design down to the module level. The subsequent section analysis the verification process of the HPDP design and shows which test will be applied in manufacturing. After that, a discussion about additional tests for commissioning is given. The fourth section starts with an introduction of different approaches in hardware testing along with a short review of the literature. After introducing the test setup, the detailed concept including test flow and individual test cases is described. Next, the implementation section explains the realized of on-board

tests into detail. Afterwards, the next section shows how the commissioning tests are verified, and presents the results. Finally, a conclusion summarizes the work and addresses open issues.

1.3 Software used throughout this Thesis

The commissioning method is implemented using two different development kids. First, the PACT XPP SDK provides several tools for compiling, simulating, visualizing, and debugging source code for the XPP-Core of the new array processor. Second, a ModelSim testbench provides a more detailed and comprehensive simulation model of the HPDP design.

2 High Performance Data Processor (HPDP) architecture

The target applications of the High Performance Data Processor (HPDP) are for high data volumes in the signal-processing domain. The flexible and in-orbit reconfigurable core is fully programmable and provides a full range of processing capabilities. The HPDP includes the eXtreme Processing Platform (XPP) Intellectual Property (IP) from PACT that combines a coarse-grained reconfigurable dataflow array with sequential processing units. Additional, space-oriented modules such as data transfer and memory interfaces are added by Astrium. [PAC09a]

The final semiconductor technology for the HPDP is 65 nm wide and radiation-hardened. The planned operational clock frequency is 250 MHz due to the space conditions. This chapter gives an overview of the processor design and its functionality from the top level design down to the module layer.

2.1 Top Level Design

The HPDP chip comprises the XPP IP core along with useful peripherals. The XPP-Core is based on the 40.16.2 XPP Dataflow Array and two Functional Processor Array Units (FNC-PAE), surrounded by different memory access and streaming objects, as well as a packet-oriented communication network. [PAC09h]

The high-speed interfaces SpaceWire with three links and the Stream-IO block with four channels offer access for controlling and data transfer with other board elements such as additional HPDP chips or instruments. The System Controller manages the different clock frequencies of the chip and peripherals. It also has a JTAG (Joint Test Action Group) interface and different operation modes for testing and debug purposes. The XPP Memory Arbiter provides interfaces for three memory ports. The first port can connect the chip to an EEPROM or SRAM, mainly to store instruction code. The second port is suitable for a large SRAM or SDRAM storage, and the third interface already connects an internal 4 Mbyte SRAM. Further, a watchdog module observes a program execution and throws an interrupt if a program does not reset the watchdog timer in a chosen period. [SH10] The Top Level Design is represented in figure 2.1, where the blue area shows the XPP part that is explained in more detail in section 2.2. The following section describes the peripheral components represented in the white area of figure 2.1.

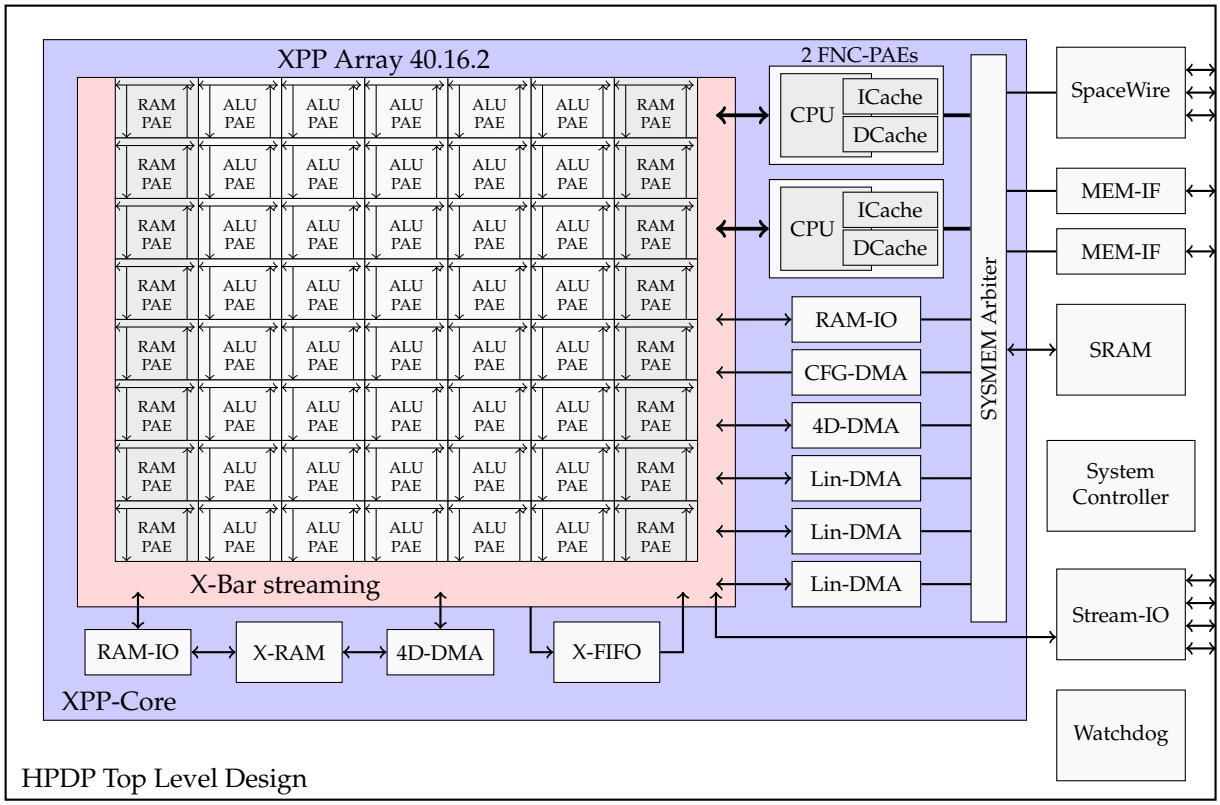


Figure 2.1: Overview of the HPDP architecture; the blue area contains the XPP-Core, the white part additional pheripherals; the arrows show the directions of data and event flow

2.2 XPP-Core

The integrated XPP-Core comprises the XPP Array, two FNCs, storage components, various memory access modules, as well as a communication network with point to point or point to multi-point connections. The dataflow array consists of two different Processor Array Elements, ALU-PAEs and RAM-PAEs, and runtime-configurable communication buses. The operational fields are high-bandwidth stream-based applications. Two FNC-PAE are coupled to the array's communication channels and perform high-performance execution of irregular and control-dominated algorithms. They are also in charge of the system management, i.e., control DMA operation. Various DMA controllers transfer data between external memories and the XPP components. Integrated FIFOs uncouple the DMA channels from the system memory bursts and potentially stalled pipelines within the XPP core. At runtime, configurable X-Bar switches connect the streaming input and output ports of on-chip components. The SYSMEM-Arbiter handles concurrent RAM ac-

cesses of various memory channels from the XPP side to the three memory ports, or vice versa. [PAC]

2.2.1 XPP Dataflow Array

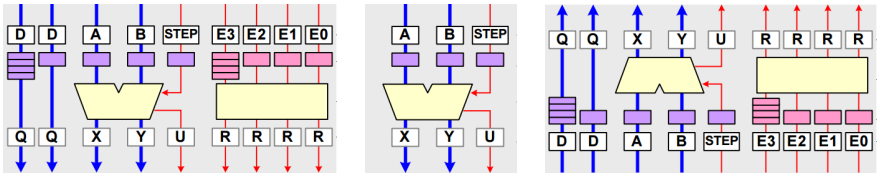
The dataflow array has about 150 16 bit-wide ALUs and 16 small 512 Mbyte SRAMs and is fully reprogrammable by the Config-DMA. Vertical routing registers and horizontal routing buses connect the components via 16-bit data and 1-bit event connections. I/O objects on the left and right sides offer functionality for the link with other XPP components via crossbars. Thereby, a ready/acknowledge protocol for communication is employed guaranteeing self-synchronization. [PAC09g]
Each array unit performs its programmed operation as soon as all required data or event input packets are available. Afterwards, the ALU forwards the result once it is computed. Hardware protocols handle pipeline stalls automatically. During a clock cycle, an XPP Dataflow object consumes data and event packets from its input ports, performs the configured operation and releases the result on the output ports if the previous packet has been consumed. If not all required input ports for the set operation are covered by a packet, the computation unit does not perform any action. Further, there are two different Processor Array Elements: Arithmetic Logic Unit PAEs (ALU-PAE) and Random Access Memory PAEs with I/O (RAM-PAE with I/O). [PAC09h]

ALU-PAE

An ALU-PAE, shown in figure 2.2, consists of three different objects that can execute a separate opcode from a limited list of operations. The ALU-object is located in the middle and performs arithmetical and boolean operations, as well as comparisons. The Forward Register-object (FREG) and Backward Register-object (BREG) offer operations for flow control, data manipulation, counters and shift operations, and are in charge of vertical routing. Additionally, FREG and BREG objects have two input FIFOs with one and four stages each to support pipeline balancing. During the configuration phase, preloading of values into FIFOs is possible. The output registers can store one value to guarantee that pipeline stalls do not lead to packet loss. [PAC09g]

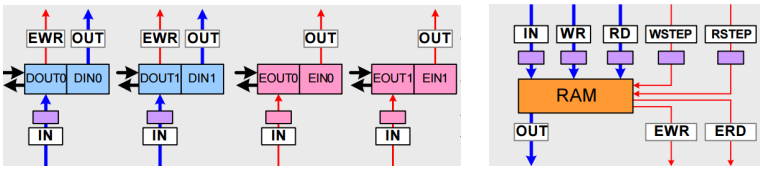
RAM-PAE

The left- and the rightmost column of the Data-flow Array consists of RAM-PAE objects. Those have one internal RAM object, one I/O object, and also an FREG and BREG object for vertical routing. The I/O object provides the ability for direct communication of the Array with other XPP components such as the FNCs or memory access objects. Both



(a) FREG (b) ALU (c) BREG

Figure 2.2: The three objects of an ALU PAE: Forward Register-, ALU- and Backward register objects with input FIFOs and funtional units (red: event channels, blue: data channels)[PAC09g]



(a) IO (b) RAM

Figure 2.3: The objects of an RAM PAE: I/O-object with streaming ports and input FIFOs, RAM-object with interal memory and input FIFOs[PAC09g]

data and events can be transferred between the ports of an I/O object and the connected network ports.
The internal RAM object works as a cache for the other array elements. It is not directly accessible from the outside but can be filled with arbitrary values at configuration time. Each RAM object can store up to 512 Mbytes and a RAM mode as well as a FIFO mode are available. Figure 2.3 depicts the RAM-PAE object. [PAC09g]

Bus Switch Object

The XPP Array includes unidirectional data and event buses for horizontal routing. Bus switches have a delay of one register and also includes one FIFO stage in their input ports. [PAC09h]

Data and Event Streams

In the XPP architecture, single data and event packets travel through the flow graph sequentially. Thereby, 1-bit events may be used for synchronizing external devices.
All array components work with self-synchronization which means that an output packet is only generated when all required input data and event packets are available. That simplifies programming and compiling of dataflow-oriented algorithms because only the

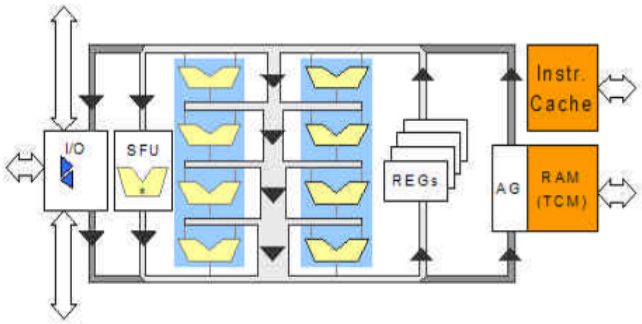


Figure 2.4: A FNC-PAE includes eight ALUs arranged in two colums, a Special Functional Unit (SFU), 16 bit registers, 32-bit address generator (AG) and a L1 Instruction cache and data cache [PAC09g]

number and order of packets travelling through a graph representation is important rather than the timing of the pipeline. [PAC09g]

2.2.2 FNC-PAE

The HPDP architecture has two Function Processing Array Elements (FNC-PAE) that are sequential 16-bits Harvard architecture processors and comprises a design similar to the Very Long Instruction Word (VLIW). The differences are an implicit conditional operation and sequential and parallel execution of eight ALUs in one clock cycle. Four ALU elements are arranged in one data path so that four ALUs can be chained. Also, predicted execution can be implemented. [PAC09a]

Therefore, an FNC-PAE is specialised for typical signal processing algorithms and for housekeeping tasks such as programming DMA components or reconfiguring the XPP Array. An FNC-ELF-GCC compiler provides the ability to execute legacy C-code. Each FNC-PAE includes L1 data and instruction cache, whereby the L1 D-cache can be used as tightly-coupled memory (TCM) alternatively. Further, a Special Functional Unit (SFU) for executing instructions like multiplication is available. [PAC09g]

2.2.3 Memory Access

Three different direct memory access (DMA) types can transfer data streams between XPP components and the System Memory Arbiter (SYSMEM Arbiter) that forwards the data to the different memory ports.

SYSMEM Arbiter

The System Arbiter processes all memory read and write requests from XPP Core components such as FNC-PAE, Linear DMA, and 4D-DMA to the corresponding memory outputs. The arbiter routes the requests independently, is fully pipelined and supports burst memory access and programmable prioritization. [PAC09h]

Linear DMA

The HPDP has three Linear DMA controllers that provide data stream accesses from the core to the memories connected to SYSMEM Arbiter, or vice versa. It automatically generates linear increasing addresses from a given address range and converts a 16-bit input stream into 64 bit-wide sequence for write access. Likewise, a memory read access is transformed into an output stream. [PAC09h]

4D-DMA

If a more complex, multi-dimensional address pattern is required, the 4D-DMA controller can be used to generate 4-dimensional address patterns. The HPDP owns four independent 4D-DMA modules for read and write access. Each 4D-DMA operates in one direction and has various modes from single 16-bit streaming until using the full 64-bit of a memory channel. [PAC09g]

RAM-IO

The RAM-IO object is a custom-built model that provides arbitrary access pattern for unrestricted 32-bit memory access between the XPP Array and external RAMs because a Dataflow Array element can only deal with 16 bit. Three data ports and one event port are available, and integrated FIFOs and buffers provide fully pipelined access to memories. [PAC09b]

2.2.4 Other components

X-RAM

The X-RAM is a small local buffer memory that is only accessible from the dataflow array via write 4D-DMA-L0 controller or the RAMIO-L object. It provides 8192 words of 16 bit each. [PAC09g]

X-FIFO

The X-FIFO module can be used as a buffer for any XPP 16-bit data stream. It uncouples a data source from a target of a stream-based processing chain, for example, to avoid stalling in data exchange between an FNC-PAE and the XPP-Array. [PAC09c]

X-Bars - Network on Chip

The XPP communication includes data streams, event streams and memory streams. The X-Bar components build the routing network on the XPP-Core and routes data or events from one input to one or more outputs. The data flow is automatically synchronized, and up to four packets can be stored if the pipeline stalls. Therefore, a component consumes a packet only if all required input packets are available, and the result is only forwarded if the previous results have been consumed. The transmitting system transfers one packet by cycle, and no packets can get lost during a pipeline stall or re-configurations process. [PAC09e]

2.3 Periphery

The Periphery extends the XPP IP core with interfaces for communication, data transfer and external memories to customize the array processor for the target area in space applications.

2.3.1 SpaceWire

The standardized SpaceWire is a spacecraft communication network specified by ESA (European Space Agency). It is used for controlling and monitoring the HPDP chip, as well as loading new configurations or programs into RAM. The SpaceWire module consists of three links and implements RMAP (Remote Memory Access Protocol) for path address routing and limited logical address routing from one SpaceWire link to another. That routing capability allows a large number of HPDP chips to interface with a single SpaceWire port on the command & control module. After chip reset, all three links are in autostart mode so that a start configuration for activating a link is not necessary. The target data rate of the packet transport is 200 Mbits/s. [Lem10]

2.3.2 Stream-IO Block

The Stream-IO module of the HPDP chip is a 16-bit wide, point-to-point bidirectional data stream interface. It provides four high-speed data communication channel links with

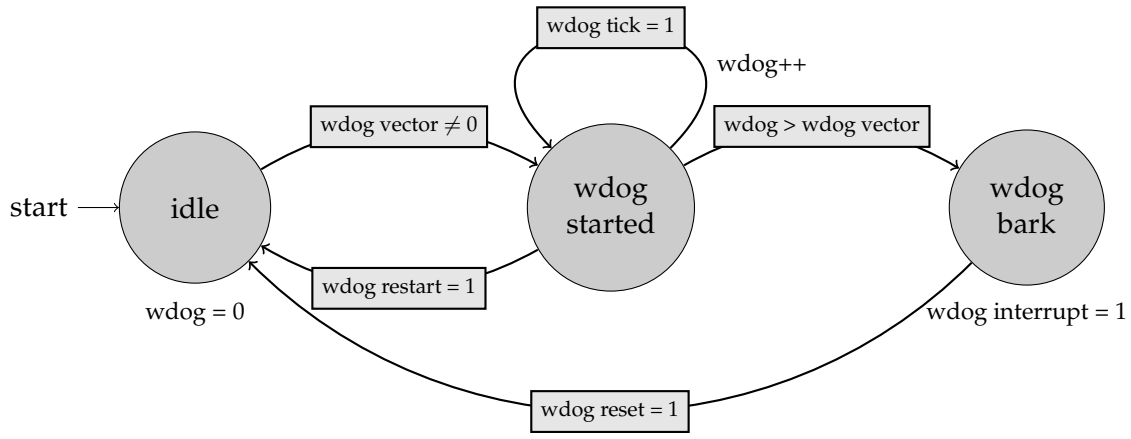


Figure 2.5: The state machine of the watchdog module

external devices, such as other HPDP chips or FPGAs, or on-board instruments. Additionally, a simple hardware protocol provides full handshake control, and different clock domains for both communication sides are possible. [PAC09d]

2.3.3 Memory Ports

The HPDP architecture includes three memory ports. All interfaces are connected with the SYSMEM Arbiter that provides a 64-bit memory channel interface. All external memories provide an EDAC (Error Detection and Correction) based on Hamming Code for protection against radiation effect. Port 0 is used as configuration memory and can be connected to a PROM/EEPROM or SRAM device storing boot code or application code. Its address space is divided into two parts. The lower address space belongs to the external memory interface, and the higher one targets the AHB master bridge to control various HPDP registers such as SpaceWire or Watchdog. Therefore, it is possible to send data from memory port 0 to external devices via SpaceWire. Memory port 1 can link either an SRAM or SDRAM memory. Port 2 is already connected to an integrates 4-Mbyte SRAM module. [Lem10] Appendix A.1 contains a tabulated overview of the connected memories (table A.1) and other technical data of the HPDP chip.

2.3.4 Watchdog

A watchdog is a necessary module for fault detection in space equipment. It is used to detect failures that affect cyclic program execution by the HPDP. Whenever the time elapses, an interrupt is set to detect the error by the controller [Skl76].

Figure 2.5 depicts the state machine of the watchdog module. If the wdog vector is unequal zero, the watchdog automatically starts and increases the value of wdog at every clock cycle by 1. When the value is higher than the programmed vector, the state machine switches to the wdog bark state where an interrupt is triggered. Afterwards, the wdog resets by itself. The wdog vector is divided into two registers, a prescaler and a value, to provide 32-bits since 16 bits would not be enough [Lem10]

2.3.5 GPIO

The GPIO (General-purpose input/output) provides eight pins for input and output signals those can be written or read by the FNC-IO-BUS. The GPIO should be used for simple input and output signals under FNC-PAE software control. Particularly, an input signal can be used to read configuration switches or to capture the status of external devices. An output pin is useful to steer debug LEDs or control external devices. [PAC08]

2.3.6 System Controller

The HPDP System Controller includes several PLLs (Phase-locked loop) that are responsible for the input clock generation of the XPP array, FNC-PAEs, SpaceWire, memory controller, and Steam-IO. All clocks are synchronous and can be adjusted by a divider and multiplier. Furthermore, the System Controller contains a debug controller and JTAG interface. [Bau15]

3 Design analysis

The HPDP development started around ten years ago. Several design verification methods and techniques have been applied on the functional level and hardware level. This chapter analyses what has been done, followed by a discussion of what additional tests should be done to bring the new array processor technology into operation.

3.1 Previous design verification

The complex System-on-Chip (SoC) design consists of an enormous number of modules, different bus systems, and memories. Thereby, two design methodologies characterise the chip due to historical reasons. That follows a proper verification strategy is needed to guarantee the correct function of the HPDP chip [LS10].
The following subsections give an overview of tests performed on the HPDP chip. The outline based on the documents [LS10] and [PAC09f].

3.1.1 Verification by PACT

The XPP-III IP core of the HPDP chip was developed by PACT, which has created integrated test suites and test benches during the design process. That includes functionality as well as top-level design tests. Moreover, all XPP-III components are co-designed in Verilog and System-C. A Design Checker reviews the Verilog code. The test benches are self-checking and can be used for RTL and gate level simulations.
Besides small component tests, application tests have shown that the processor architecture is capable of executing relevant applications using the full ability of XPP core. Examples are a JPEG encoder, FIR filter or Fast Fourier transform.

The FNC-PAE is the most complex module of the HPDP architecture. Therefore, own sample applications have been created. Both an FNC Software Simulator *xfncsim* as well as an RTL simulator for the Verilog specification, execute test cases to prove functional correctness. These two test frameworks generate the same kind of trace files that contain detailed information about the internal state of an FNC-PAE in every clock cycle. These trace files are useful to prove the equality of both test results since the Verilog and SystemC code are based on the same HPDP specification. Besides, line coverage of the Verilog code is analysed using hardware development tools.

However, the simulators have a different cycle accuracy due to the complexity of some modules like L1-caches or hardware dependencies. The simulation language SystemC is not able to simulate an asynchronous clock domain that is located between the FNC-Container and the system memory arbiter. The simulation uses a simplified model that mocks a corresponding number of cycles for a cross-domain simulation.

An additional SystemC simulation tests the XPP Array and the interacting modules for networking and direct memory access on the system level. The same trace file strategy as for the FNCs is used.
Further, simple and complex access patterns verify the memory interfaces. The Stream-IO block is also provided by PACT in an extra module with an own test suite because it is not part of the actual XPP IP core.

3.1.2 Verification of complete HPDP design

The final version of HPDP combines the PACT modules XPP-III 40.16.2 core and Stream-IO block, and supplemental modules such as a SpaceWire module, three memory ports, a System Controller, and a JTAG Controller.

Verification tests consist of several steps that are performed on a module, a sub-system and at top-level. Those tests ensure that the HPDP chip is operating correctly, according to its specification.
In the first step, the complete integration of Astrium-modules is verified at the design and subsystem levels. Besides, test cases check interconnections and module overlapping functionalities such as interrupts, read and write on memories and registers.
The second step contains integration of the PACT testbench in the Astrium developed test benches to simplify the integration of Astrium modules into PACT-IP.System level functionalities, top-level modules, and pad interconnections are tested.
To achieve a sufficient design coverage, test generation application (TGA) generate a couple of tests automatically. Several of those TGAs are contained in the verification environment and create multiple tests to cover different fault classes of a certain feature. Moreover, single XPP components are tested with an extensive regression regression test suite.
Included application tests have been written to nearly cover all modules of the top-level design. Several component tests use connectivity modules of the top-level design such as crossbars and memory arbiters in passing. Therefore, those components are also assumed to be fault-free.

Two memory ports connect an external data carrier with the HPDP chip. Those memory blocks come from external providers, and it may assume that those are well tested.
Hardware must resist certain physical conditions. Primarily space equipment is exposed to high and very low temperatures during operation and idle state. The maximal power consumption and the consequent cooling demand can be calculated with tools, and the chip vendor will guarantee the estimated thresholds.

3.1.3 Manufacturing tests

Two steps of equivalence checking will be done during the implementation process. First, Astrium proved the equivalence between the RTL code and the netlist transferred to the Backend Design house using formal verification techniques. Second, equivalence checks between the transferred and the final produced netlist will be performed before delivery. These controls shall guarantee a proper functionality of final hardware relation to design.

The HPDP netlist is designed by using the design for testability (DFT) technique which means that testing features are added to the chip simplifying developing and applying manufacturing tests to the designed hardware. Therefore, the processor contains scan-chains, boundary scans, and two additional debug modes next to the regular operation mode. That offers the option to observe and test every flip-flop in debug mode, included read and write to internal RAMs or FNC instructions.
A BIST (Built-in self-test) logic and the corresponding BIST mode verify the correctness of each internal RAM cell by applying different test patterns created by an ATPG (Automatic Test Pattern Generator). [Lie10]
Further, static timing analysis (STA) will be done following the rules from the chip vendor. It contains best and worst case analysis in typical modes. STA is capable of checking every structural path and can detect glitches, slow parts and clock skews using scan shifts, stuck-at capture and delay capture. Power Distribution, Voltage (IR) Drop and Electro-migration on chip shall be tested, as well as observation of the PLL performance. [Bau15]

3.2 Discussion

The HPDP chip is based on a new technology of array processors that is not well established yet. Therefore, carefully testing of the semiconductor could be necessary after manufacturing. As the previous section has shown, many tests are already done before commissioning. The design is thoroughly verified using various tools and testbenches; the circuit is designed for testability (DFT) and post-production tests as boundary scans and delay tests will be applied. Consequently, the assumption is that the manufactured

Phase	Part	Verification
A. Design by PACT	XPP-Core design	Verilog design tests
		SystemC simulations
	Stream-IO Block	Verilog design tests
B. Extension by Astrium	HPDP design (XPP-Core + periphery)	VHDL design tests with ModelSim
C. Manufacturing	Pre-production netlist	Equivalence checking between RTL design and netlist
	Post-production	Various random vector tests on ASIC tester
Open issues for commissioning	→ Stress test of whole HPDP chip in operational mode	
	→ Functional connectivity of internal and external memories	
	→ Functionality of SpaceWire module	
	→ Data transfer via Stream-IO ports	
	→ Operability of the Watchdog timer	

Table 3.1: An overview about the different design phases and open issues. Each phase is divided into the central parts and tests those are done or are going to happen before delivery.

chip comprises the expected design and should be free of hardware faults.

However, additional tests on the fabricated HPDP chip are required to ensure the compliance of realised technology, because the simulations cannot fully guarantee the intended hardware behaviour of the manufactured semiconductor, and the manufacturing tests do not run software programs. Thus, the regular interaction of written software and the designed HPDP architecture must still be demonstrated on real hardware. The next subsections analyse and discuss the available software tools for developing, as well as internal features and interfaces those are interesting for commissioning tests. Table 3.1 gives an overview of the analysis.

3.2.1 Drawbacks of software simulations

Various SystemC simulations and a given ModelSim testbenches can simulate the complex HPDP design and prove its correctness. However, those have different disadvantages regarding the limited scope and simulation time.
Due to the nature of the system-level of SystemC tools, the simulations for the XPP core

or only the FNC-PAE are quite fast and can simulate millions of clock cycles in a reasonable amount of time. Some modules, like the SpaceWire, have been added later and are not included, which limits the SystemC simulations and makes it impossible to develop a whole commissioning test suite. Also, the system-level simulations simplify the steps for computing but prevent the simulation of correct timing. In particular, the timing of Clock Domain Crossings, FNC caches, and external memories are not represented correctly [LS10].
The ModelSim tool follows a different approach simulating Verilog or VHDL code at the register-transfer (RT) level. Consequently, the simulation is very close to the real hardware and details are much better represented than a SystemC simulation does. However, the enormous complexity of the HPDP design makes an RT-level simulation very costly in runtime and required memory. Hence, a simulation time of a few milliseconds can take several hours. Although all components of the final HPDP chip are present in the ModelSim testbench, testing of complex algorithms is very time-consuming for fast prototyping of commissioning tests.
As an example, it is acceptable to verify the watchdog with a small timer so that the watchdog interrupt is triggered after a short simulation time. When the timer, however, adds up to a couple of seconds to trigger, in the case of testing also high values, it would take days of simulation only for one test. Of course, the design is standardised, and if the logic works for milliseconds with a complete code coverage, it should also work for seconds or minutes. However, the hardware is complex and must also be proven if that is the case. Besides, the watchdog timer is a peripheral module and is not included in the XPP-Core.

3.2.2 Testing HPDP interfaces

Using the HPDP chip requires a board that provides ports and slots for SpaceWire, Channel links (Stream-IO), GPIO, and memories. As the planned board and processor chip are manufactured independently, the connections between the board and chip are not verified before commissioning.
Thus, an important part of the commissioning testing is the verification of the reliability of all interfaces. Chapter 4 introduces the test setup and describes the available interfaces in detail.

3.2.3 Accessibility of memory modules

The planned board provides two board memories which utilise the HPDP for data and program storage. Further, the processor chip contains several internal RAMs and caches. With the design for testability (DFT), each memory has a Built-in-Self-Test (BIST) that is

an approved standard and checks every memory cell with different addresses and value patterns. Moreover, external memories can be assumed to be well tested by suppliers.

However, the functional accessibility of each memory by the computation units remains an open question. Data processing validation must show that each memory is probably connected and addressable by functional units or the SYSMEM arbiter. For this, a test may apply different input vectors such as all-zero pattern, all-one-pattern, random, or counter pattern. Further, L1 data cache and instruction cache of FNC-PAEs can not be directly addressed by C-Code or Assembler (only a half of L1-D-Cache is usable as Tightly Coupled Memory (TCM) [PAC]. Hence, FNC stress tests are also required to reach a good test coverage of FNC memories.

3.2.4 Clock-domain crossings and power supply

The HPDP is expected to have an operational speed of 250 MHz. That clock frequency applies to the central part of the XPP-Core, which includes the XPP-Array and the memory access components. The FNC-container, in turn, runs on a unique clock domain, which has half of the XPP-Core's operational speed, i.e. 125 MHz. All data transfer modules need to deal with the frequency of the channel connected to the HPDP. Therefore, clock-domain crossings are included whenever two clock domain meet. There are also several clock domains inside the chip, since planned clock frequencies for external memories, SpaceWire and Stream-IO may less than the HPDP reference clock. Delay faults make those clock-domain crossings vulnerable to metastability, data loss, and data incoherency. Although the manufacturing tests should be quite comprehensive, their effectiveness under operational conditions is questionable [KCGP12].

For that reason, stress tests are suitable to determine the stability of clock-domain crossings and power distributions under operational conditions. The higher the processor frequency, the more vulnerable the hardware is for such defects [CDS+00]. That implies that component tests should also be done on a real HPDP to guarantee a correct behaviour of internal features and interfaces under full load.

A software program is suitable for stress testing since instructions utilise the entire chip rather than only a few units. Consequently, comprehensive stress tests should be done running with a maximum number of functions and interfaces to discover potential side-effects on the power supply and establish the proper integration of critical components. [KLC+02]

3.2.5 XPP Array test

An array processor architecture gets more and more into focus as it has advantages in the stream-based processing domain over traditional DSPs or SIMD processors. This processing part of the HPDP is located in the XPP IP core, and various testbenches have performed cycle-accurate tests on system-level as well as RT-level. Further, the entire array runs in one clock domain and does not utilise any chip interface. The comprised RAM-PAEs has a BIST structure, and the netlist of the array has several scan-chains. Thus, the XPP Array has been extensively tested and a complete verification program that checks every opcode and connection is not necessary. A simple test which shows the basic functionalities, as well as a memory test for the included RAM-PAEs, should be sufficient.

3.2.6 Components tests

Even though components such as an ALU-PAE or DMA controller are well verified before commissioning, it does not mean that it works in every case. Assumed that the worst case will happen, and a component does not operate in a case where an FNC program executes specific instructions, an agile test methodology would show the exact fault function after running a particular set of tests. However, the HPDP design is intricate, and providing detailed test cases for every functionality goes beyond the scope of this work. Instead, this thesis focuses on the development of a test suite that covers all critical features assuming that non-critical features work as expected. Together, the test cases include all components, targeting each of them at least once. The JTAG interface can be used in combination to investigate a occurred fault in more detail.

4 Test concept

The purpose of this work is to develop an on-board test environment software that runs on the new HPDP chip. In particular, critical internal features and interfaces of the array processor shall be tested. As the analysis has shown, several test suites and simulation tools already exist, and some are suitable for planning the commissioning tests. However, a real HPDP demonstrator chip is not available so that individual test programs cannot be verified on manufactured semiconductor.

This chapter starts with an introduction of hardware testing approaches and detailing the preferred software-based self-testing methodology. Subsequently, the test setup for commissioning the HPDP chip in the laboratory is presented. Finally, a proper test concept is discussed in detail by explaining the order of required tests as well as the steps of every single test.

4.1 Test approaches in Processor Testing

Advanced technologies, increasing clock frequencies and decreasing sizes of integrated circuits have revealed new challenges for hardware testing to guarantee the reliability of devices meeting the requirements set up during the design phase. As a result, three major test approaches have been established in the testing field of processor chips to deal with testing challenges such as potential structural and functional faults.

4.1.1 External tester

The first testing strategy uses an external high-speed tester that proves the reliability of a semiconductor using scan-chains and boundary tests, for instance. Due to rapidly increasing device speed, a performance gap raised between such automated test equipment (ATE) and I/O speed of devices under test. This problem has led to over-testing and may cause many false positives, resulting in yield loss [KLC+02, BCD03].

Consequently, self-testing, the ability of a circuit to test itself by generating the required test patterns on-chip and applying the tests at-speed, has become into focus and eliminates the need for high-speed testers [CDS+00].

4.1.2 Built-in self-test (BIST)

A well researched self-testing methodology is the built-in self-test (BIST), that uses an embedded hardware test generator and test response analyser to generate and apply test patterns on-chip and at the operational speed of the circuit under test.

Structural BIST, such as scan-based BIST, have an acceptable test quality but cause a performance and design time overhead for the additional dedicated test circuit. Also, a structural BIST consumes much more power than normal system operation which arises new complex timing issues in areas like multiple clock domains and can stress the device additively [KLC+02]. However, BIST is useful in components such as integrated memories and can reduce the yield loss stemming from tester accuracy problems.

As mentioned before, all internal RAMs have an integrated BIST structure and can be used for manufacturing tests as well as the scan-chains by the tester.

4.1.3 Embedded software-based self-testing (SBST)

The newest approach is the software-based self-testing (SBST) that was introduced to alleviate the problems of high-speed testers and BIST. SBST utilise the instruction set of a programmable core and its functionality to run an automatically synthesized test program. An on-chip pattern generator and response analyser can verify buses and interfaces. The test program is then stored either in a dedicated ROM or loaded into RAM by a low-cost tester. The approach is non-intrusive and analog, digital and mixed-signal components can be tested. [KLC+02]

SBST has several benefits over structural BIST and high-speed testers. The methodology does not need an explicit on-chip controller, what also avoid an area, performance, and power consumption overhead. Moreover, SBST is suitable for processor cores, memories or interconnections, and testing at the processor's operational speed is possible avoiding under- and over-testing.

However, there are also a few drawbacks compared to BIST. Creating test programs cost additional runtime; the intern memories must have enough capacity, and the SoC needs a programmable processing unit. [AOM06]

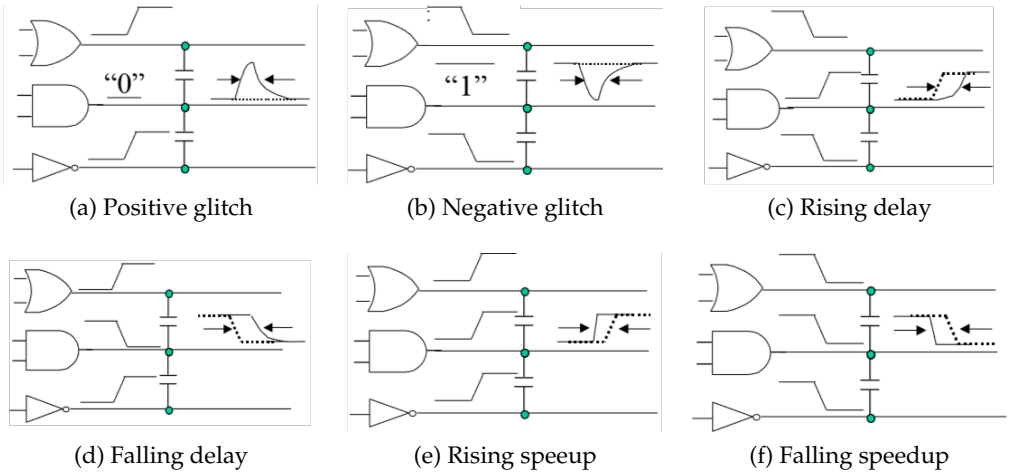
This work deals with the commissioning of a completely new designed and manufactured HPDP chip. The design can not be extended by additional structures like a hardware pattern generator. Tests which are only steered by an external tester has speed drawbacks for testing analysed tasks such as memory connectivity or stability of power supply. Thus, the intended test approach should be related to SBST.

4.1.4 Previous Work for SBST

An extensive review of different methodologies in SBST is given in [KLC+02]. The first time SBST was mentioned was by Chen at al. [CDS+00, CD01]. They use pseudorandom test pattern generation on-chip, which is motivated by testing a processor with a sequence of instruction [BP99]. The pseudorandom strategy yields to a large self-test code size and test program. The researched test core was a simple Parwan chip.
Due to the weakness of pseudorandom patterns guaranteeing a high fault coverage, Paschalis at al. introduced a deterministic test approach which targeted the ALU and shifter object [PGK+01]. Later, the researcher team published an expanded strategy targeting each component of a Parwan CPU and reduced the code size [KPGZ02, KPGZ03].
Furthermore, Chen at al. represents a scalable methodology using program templates and statistical regression method for functional mapping [CRRD03]. The same researcher team introduced a diagnostic tree featuring the splitting of one major test program into smaller ones where each program only covers a few faults [CD02]. The approach is based on a tree structure for storing a full fault dictionary introduced in [BHF96].
Based on the processor's SystemC description, Goloubeva at al. presents a mutable algorithm to generate low-level test vectors reaching a high fault coverage [GRV03]. They believe that high-level generated test vectors are not sufficient to cover all possible failures of the system.
All those introduced approaches focus on the processor core and target stuck-at faults. They do not consider other components of a SoC, such as network elements, controllers, and memories. Also, functional and stuck-at testing does not target more complex problems like target delay faults.

Lei at al. present a self-testing methodology for path delay faults using processor instruction [LKC00]. Their conclusion is that delay faults in none-functional testable paths will not affect the chip and do not lead to failure. The set of functionally testable paths is a subset of all structurally testable paths, and a none-functionally testable path is never performed in normal operation mode. Singh et al. come to the same conclusion by using a finite state machine of the controller to find path delay faults in functionally testable paths [SISF05].
Bai at al. describe a more advanced approach where the authors tackle crosstalk faults which can only be detected by sequences of test vectors at operational speed [BCD03]. Figure 4.1 gives an overview of issues discovering crosstalk faults. Further, Benso at al. address a lack of focus on the more complex functional blocks, such as the pipelining interlock mechanism or the cache hierarchy. Those components can not be ignored while testing microprocessor's core, but are usually directly addressable by the processors in-

struction set. [BBPS06]



(a) Positive glitch (b) Negative glitch (c) Rising delay

(d) Falling delay (e) Rising speeup (f) Falling speedup

Figure 4.1: Possible crosstalk faults in integrated circuits which must be tackeled at operational speed of the processor [BCD03]

Discussion

Altogether much work has been done in SBST for single-core, multi-core, pipeline processors and bus systems. Nonetheless, the literature does not treat with the recent array architecture like the HPDP has. Also, the publications are concentrated on manufacturing tests. According to the technical documents [LS10, Lie10, Bau15], the FNC instruction set is not used to discover functional paths or crosstalk faults. Hence, a high need for commissioning programs is required.

However, the HPDP owns a very complex architecture, and all components are configurable by the FNC instruction set or configuration registers directly programmable. More precisely, the FNC has over 100 different instructions, the processor units of the XPP Array more 27 other instructions and the periphery modules such as SpaceWire, MemoryPorts, Watchdog, and Stream-IO have more than 800 controllable registers altogether [PAC09g, Lem10]. Also, the previous analysis has shown that a complete test is not mandatory to achieve the objective of commissioning.

Hence, the test concept differs from the introduced software-base self-testing methodology represented in the literature. On the one hand, the instruction set of the processing units shall be used to cover complex hardware problems which occur only at maximal operational speed, and on the other, not each instruction and especially not all combinations of instructions have to be included in commissioning testing.

4.2 Test setup

An HPDP chip must be placed on the main board, which provides ports and slots for interfaces and external memories. For test purposes, the board needs further connections to a master tester to send commands the HPDP then. Even though a real HPDP demonstrator chip is not supplied during the implementation, the commissioning tests must regard the provided test setup.
Figure 4.2 depicts the test setup including a test system, a board, and an HPDP chip. The test system runs on a Board Level Test System (BLTS) that is explained in the following subsection. The depicted board is a multi-chip board that has slots for up to four HPDP chips. Both, single-chip and multi-chip board are in development, and they are described afterwards.

4.2.1 Board Level Test System (BLTS)

An HPDP chip will be tested with the provided Board Level Test System (BLTS) that includes an off-the-shelf industrial computer for manual testing and automated test execution. The HPDP BLTS is designed for testing maximum data throughput of the HPDP module. Required software and hardware modules are installed. A front-end application manages the communication between the HPDP board and a UMDS Runtime System (Universal Data Management System). UDMS is a generic database driven tool and approved in many space test system projects. The off-the-shelf industry PC has an Intel Core i5-4570 CPU with 16 GBytes RAM. [Bro15]

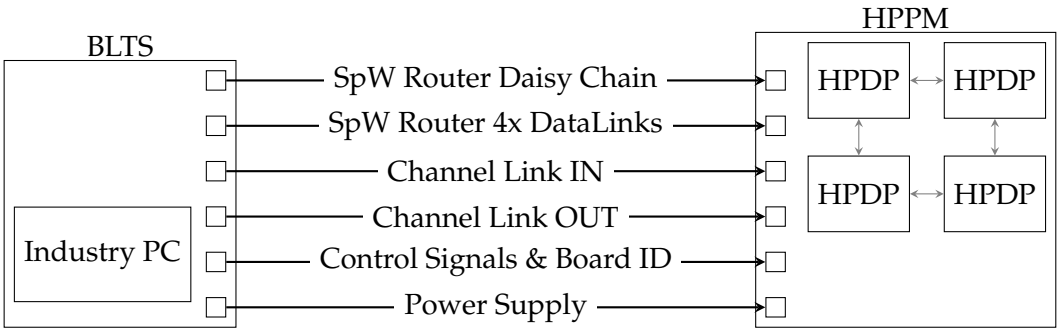


Figure 4.2: A simplified representation of the HPDP-BLTS (Board Level Test System) environment including the High Performance Processing Modul (HPPM) with four HPDP chips. [Bro15]

With such a test system, a manual command or even an automated test script can read or write, if possible, every accessible register of the HPDP via the Daisy Chain and the HPDP SpaceWire module. Thus, the system can configure the HPDP chip, copy a program into

code memory, test specific functions by static access, start the boot process, or monitor a whole program execution only via the SpaceWire connection.
Further, high-speed SpaceWire DataLinks and Channel Links are available to simulate the data transfer of on-board instruments. Each of the Channel Link IN and OUT utilises one Stream-IO port. Also, a board has an ID that is readable from GPIO pins by the HPDP chip. The BLTS environment can assign every board a unique ID as board chaining is planned in future.

4.2.2 Single-chip Board

The single-chip board is the preferred one for commissioning of the first delivered HPDP chip. It offers all necessary ports for the HPDP interfaces and contains two fixed board memories. At port 0 is an 1 Mbyte EEPROM interconnected and should mainly be used as code memory since the write speed is slower than SRAM or SDRAM. The latter memory type is linked to memory port 1 and provides 512 Mbyte storage capacity for the HPDP chip.
Besides the SpaceWire Daisy Chain, data can also be transferred from and to the board by a SpaceWire DataLink, or by a ChannelLink IN and ChannelLink OUT. Thereby, only the Stream-IO ports 0 and 2 are used. The other two ports are interconnected to a loop back. That means that data send to Stream-IO port 1 is transmitted to port 3 or vice versa.
Moreover, the single-chip board utilises six GPIO (General Purpose Input/Output) pins of the HPDP chip. The first two pins are used to activate the ChannelLink or SpaceWire DataLink connections, and the next three pins can steer three LED lights. All of them are used as output by the HPDP. The last pin is an input pin for reading the board ID bitwise. Appendix A.1 contains a tabulated overview of the technical data of the HPDP chip.

4.2.3 Multi-chip Board (HPPM)

Up to four HPDP chips can be placed on a multi-chip board called High Performance Processing Module (HPPM). All four chips are connected to a ring structure using two Stream-IO ports. Another Stream-IO port of each chip can be linked to another HPPM, and the fourth port is connected to a Channel Link. Further, the GPIO assignment and both board memories are equal to the single-chip board.
This work comprises a commissioning test software for the first demonstrator chip, and the single-chip board is firstly delivered and sufficient. Multi-chip programs are out of scope, and the tests shall primary be designed for one processor chip. Nevertheless, the operational capability for multi-chip testing is shortly discussed in section 5.4.

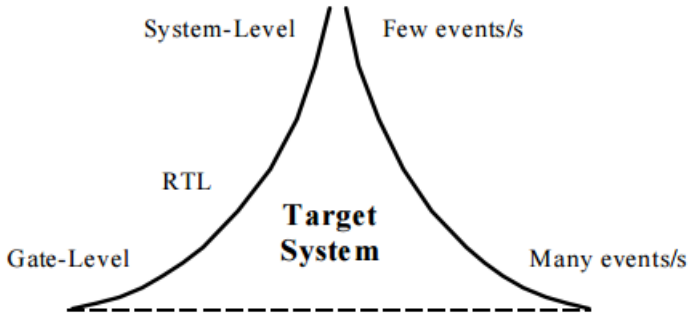


Figure 4.3: Events per second in a target system for different abstraction layers [ESL01]

4.3 Test strategy

The beginning of this chapter introduced different approaches in processor testing. The conclusion is that one essential characteristic of a test program is running at operational speed since sensitive timing affects only arise when the chip runs under that condition. SBST is a proper approach to reach that goal.
Another advantage of system-level testing, like SBST does, illustrates the figure 3. Due to the high abstraction level, fewer visible events occurs in a second what motivates a top-down testing strategy. Testing at the register-transfer level, or even harder on gate level increases the number of events to handle and also the number of checks if an error has occurred. This section describes the test approach in detail.

4.3.1 Source and sink pattern

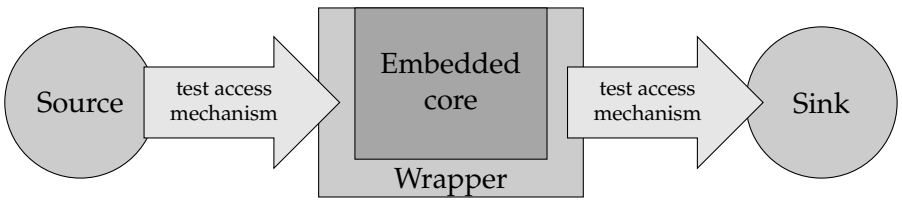


Figure 4.4: Source and sink pattern for testing an embedded core; a test sequence (or program) is transferred via a test access mechanism from a source to a wrapper that includes the core-under-test; the respond is forwarded to a sink [ZMD98]

The procedure of the SBST methodology can be nearer described by the source and sink pattern introduces in [ZMD98]. Figure 4.4 shows an architecture composed of three structural elements. The first component contains the source that generates the test patterns, and the sink, that compares the response to the expected result. The second element, the test access mechanism, transports the patterns from the source to the core-under-test

and the produced responds back to the sink. The central feature is a wrapper forming an interface between the embedded core and the test environment.

This pattern can directly be mapped to the given test setup where the source and sink represent the external tester (BLTS). The SpaceWire and Channel Links are equal to the test access mechanism, and the wrapper stands for the single-chip board (or multi-chip board). The tester loads an FNC-program into the code memory through the serial SpaceWire interface and waits for the response of the HPDP chip, sent via SpaceWire likewise. Hence, the illustrated pattern is a black box testing technique.

4.3.2 Procedure of a single test program

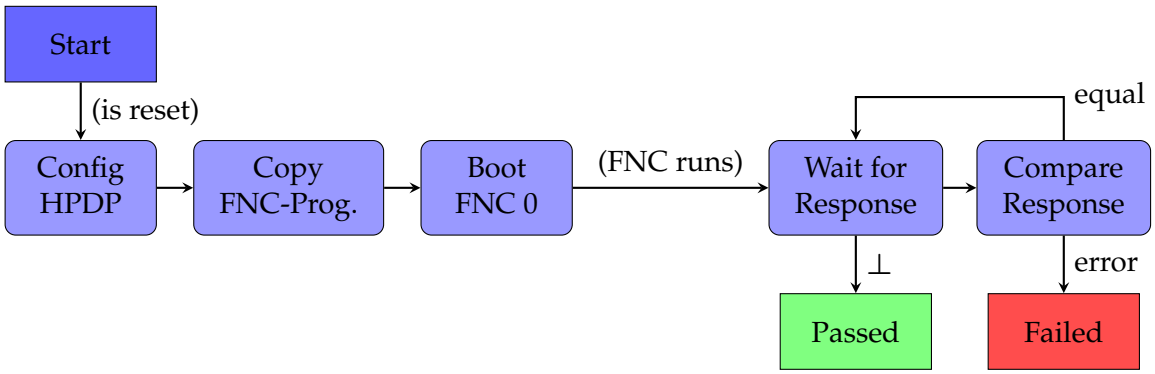


Figure 4.5: Test program flow of the script file. Each test program should start after a reset, and initialize the HPDP as required. The wait sequence for the HPDP responses follows after the boot. Finally, the results are evaluated.

The control flow of the source and sink pattern can be either executed manually by a test user or represented in a script that includes all instructions for the BLTS from the beginning until the end of a single test program. A script has several advantages compared to manual work. Repetitive and redundant work can be combined in a file to provide a common basis for simple maintainability and reusability. A typical test sequence of a script is illustrated in figure 4.5.
Every test program shall start after a reset of the HPDP chip to avoid side-effects between consecutive programs. A reset can be accomplished by disconnecting the power supply which sets all registers back to their default values. The configuration of the processor is the first step and takes a few milliseconds so that a reset should not be an obstacle. When the HPDP is in the intended state, the tester can start copying the FNC-program into the code memory and initiate the boot process. While the FNC runs, the tester (sink) waits for the responses and compares them with the expected result. A test case fails if at least

one result does not match the one expected. Otherwise, it has passed.

Figure 4.6 depicts a typical test sequence for the embedded chip. After the boot instruction reaches to the HPDP, it starts with fetching the first instructions from code memory and executing them. Those instructions usually include the default configuration actions, contained in the boot code library. Also, each FNC program includes specific configuration instructions which are required by the test case. Next, the program executes the programmed test sequence, which includes sending back one or multiple responses via SpaceWire to the tester. Finally, the system is cleaned up. That last step could be omitted since the next test starts after a reset.

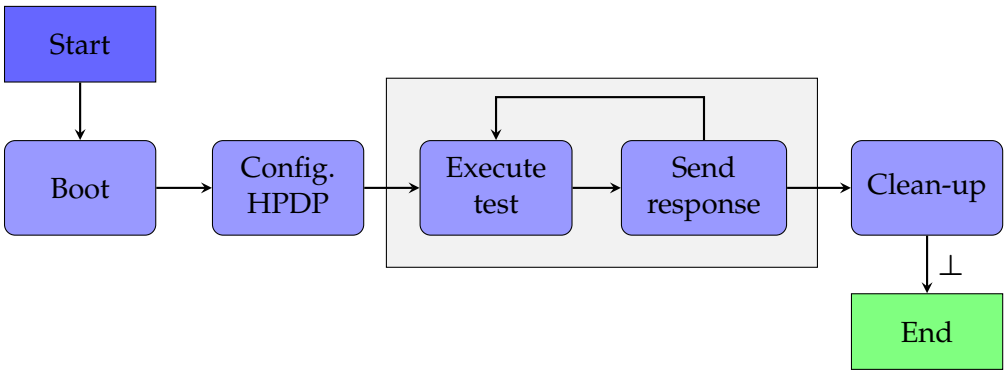


Figure 4.6: Each FNC-program starts with a configuration of the HPDP, continues with the intrinsic test sequence and clean up the system at the end. A test sequence may send messages to the external tester.

4.3.3 Sequence of several programs

A couple of test programs are required to cover all analysed, possible problem. Of course, one huge test might treat everything, but this is not beneficial for the test user regarding maintainability and reusability. In particular, some tests need to run several times with different parameters to cover everything. That concerns memory tests as test user may diverge from what is important and what is not. Besides, it does not make much sense to run two or more different tests on the same chip in parallel since side-effects or limited resources would affect the test results.

Incremental integration testing

The commissioning has to deal with a couple of problems. On the one hand, there could be HPDP specific internal issues, and on the other, an interface connection with the board

and the external tester could not work properly. Since a verification of an HPDP feature requires working interfaces, the order of tests should be planned carefully. That could simplify the classification of errors as previous tests have indicated that individual components work presumably. Big-bang testing does not make much sense here and therefore the chosen approach is inspired by incremental integration testing. In this method, every following test extends the number of tested modules with the least components as possible.

Advanced strategies

Researchers have introduced some advanced strategies like a tree structure for extracting a distinct fault from overlapping tests [BHF96]. There, each test is related to a couple of subtests which are executed if the test fails. Further, a smart tree structure with the knowledge about possible errors of each test could then be used to spot the defect accurately. As the analysis section already discussed, an advanced strategy is out of scope for commissioning and a simple sequential order is sufficient. Additional detailed tests can be performed by the JTAG interface to investigate the origin of an arisen fault.

4.3.4 Memory testing

The HPDP architecture includes several internal and external memories, supporting algorithms with different storage strategies for big data sets. Each memory contains a BIST structure for comprehensive memory tests during manufacture or running in debug mode.

Usually, a RAM module comprises a memory cell array, an address decoder circuit, and a read-write circuit. A wide-spread memory test technique finding all known faults in SRAMs is the algorithm "length 9N" (included faults are: memory cell stuck-at-1/0 faults, stuck-open faults, state transition 1-to-0 and 0-to-1 faults, state coupling faults to another cell, multiple access and wrong addressing faults, and data retention faults). [TNF01]

Even though the internal functionality of each memory seems to be sufficiently tested, the functional integration into the HPDP environment and the functional connectivity have to be verified. For instance, the interaction between the SYSMEM arbiter and a board memory address decoder is not tested by BIST and both components are separated while manufacturing. However, the SYSMEM arbiter is tested using boundary-scan-chains before commissioning. A simple test strategy looking for multiple accesses and wrong addressing faults should be sufficient. Section 5.3.1 discusses an adequate implementation.

I. Basic-functional tests	
1. GPIO test	The first test accesses the board and HPDP chip via SpaceWire and switches on the LED lights which are connected to GPIO pins. The result is observable. → Power Supply, SpaceWire 1 RMAP, GPIO (output)
2. Memories test	Verify the operational capability of the two board memories by writing values to specific addresses. Reading of the same addresses shall return the written values. → Board memories 0 and 1
3. SpaceWire test	Check the basic functionality of each SpaceWire link (1/2/3). That includes the RMAP module as well as DMA module. → SpaceWire 1/2/3 RMAP & DMA
4. Stress test	Write an enormous amount of data to memory 0, 1 and 2 via the SpaceWire links to evince their functionality under stress. → FNC-program can be copied faultless to code memory
5. Boot test	Load a small FNC-program into memory and start the boot unit FNC0. The program may activate the LEDs. → The HPDP can boot and switch on the LEDs

Table 4.1: Enumeration of basic-functional tests; more detailed descriptions are available in appendix A.3; tests of the subsequent phase are given in table 4.2.

4.4 Test phases

After analysing the verification status, clarifying the test setup, and introducing the test strategy, this section suggests an order for the incremental integration approach. Thereby, all tests are divided into two categories. The first one contains tests those should ensure that the basic functions are available or accessible. If that is the case, more exhaustive tests targeting critical internal features and interfaces can be performed in the second phase.

4.4.1 Basic-functional tests

The System-on-Chip is embedded into a board and its interfaces. The first test cases should check the responsiveness of the HPDP chip and required interfaces such as the SpaceWire link and board memories. That means that the test setup shall verify if every connection is set up correctly and if the HPDP is addressable. Only when this situation is given, more exhaustive tests should be done. Table 4.1 shows each test associated with this category.

The commissioning testing should start simple. Therefore, the first step is to program

the GPIO registers via SpaceWire so that the three board-LEDs, connected to the GPIO pins, are turned on. If that test is working, the HPDP is responsive and programmable via SpaceWire, and the power supply works.
The second test verifies all memory interfaces. The software-based self-tests are not executable if at least one of the three memories is not addressable. Therefore, that test case addresses different cells of each memory and verifies the read and write ability.
The next test focuses on the SpaceWire module and tests all three links and the SpaceWire DMA module. That SpaceWire part is required to send data back via an FNC unit. Because the SpaceWire DMA test requires the board memories, that test shall be performed after the previous memory test.
After it, all interfaces, except the Stream-IO connection, are mainly responsive. The fourth test features the possibility to stress the SpaceWire module and memories. This test should show if a larger amount of data, i.e. an FNC-program, can be written into memory without errors. Here, it is also possible to test different clock frequencies of the SpaceWire or memories, if required.
Lastly, a small FNC-program should run on the HPDP chip to demonstrate the bootability. The test may also activate the LED lights making the result more observable.

4.4.2 Functional integration tests

After verifying that the basic functions are available, further tests have to show the reliability of components such as memories and interfaces under the processor's operational speed. Moreover, the second test phase continues with the incremental integration strategy and tries to not use too many new critical components in a single test. Table 4.2 lists all constructed tests in planned order, and the following subsections discuss why choosing that order.

General tests

The second test phase starts with a simple response test verifying that an FNC can utilise the SpaceWire DMA for sending a response to the BLTS. If that functionality does not work correctly, all following tests do not perform either properly since they are all using the SPW DMA concept.
The subsequent test extends the initial GPIO test by testing the input direction at GPIO pin 5, where the board ID is bitwise available. An FNC program needs to set the output pins [4:2] to select the right bit of the board ID at pin 5. u An important test for space equipment is the verification of the watchdog timer. The watchdog is programmable via its own registers and a test with a high timer value has to be executed to show its functionality for

II Extended-functional	
G1 General	
Response test	Check that an FNC-PAE can handle a SpaceWire DMA transfer. → FNC ⇔ SpaceWire DMA
Board ID	Read the board ID bitwise from GPIO pin 5. The GPIO pins [4:2] are used to select the corresponding bit of the ID. → GPIO (input), Board ID
Watchdog	Check the Watchdog interrupt with a very high timer value. → Watchdog
G2 FNC Container	
FNC-XBAR	Transfer data and events between both FNC-PAEs via the X-BAR. → FNC 1, FNC-XBAR
FNC-TCM	Stress the entire TCM of each FNC. → FNC-TCM
FNC-Memories	Stress the memories connected to the three memory ports by writing and reading back various pattern. → Board memories 0 / 1 and internal SRAM at port 2
G3 XPP-Core	
XPP Array func.	Test the basic behaviour of the XPP Array. → XPP Array (partial), Config DMA
CDC-FNC-Array	Stress the clock-domain crossing between the both FNCs and the XPP Array by utilising all ports for a data transfer. → Clock-domain crossing: FNC ⇔ XPP Array
RAM-PAE	Verify the functional connectivity of all RAM-PAEs. → RAM-PAEs
X-RAM	Prove the accessibility of the X-RAM module. → Right and left 4D-DMAs, X-RAM.
X-FIFO	Utilise the Linear DMAs to test the X-FIFO component. → Linear DMAs, X-FIFO
G4 General II	
Stream-IO test	Exchange memory content via the Stream-IO ports. → Stream-IO
Full load test	Use as much as possible modules and interfaces at the same time. → Power Supply Voltage

Table 4.2: Overview of the chosen component, stress and memory tests. The list is in order planned to correspond the incremental integration testing. More detailed descriptions are given in appendix A.3.

an extended period.

FNC-Container

The first test of this section is only about checking the second FNC-PAE and the FNC-XBAR that connects both FNCs. That test follows the incremental integration strategy since the FNC 1 is required in the following tests.
After that, the Tightly Coupled Memory (TCM) of both FNCs can be stressed by writing and reading a big amount of data. Each TCM block has a size of 4 Kbyte, which is the half of L1 D-Cache. Hence, an exhaustive verification may also stress and verify the other half of the D-Cache, which is not directly accessible with the available instruction set.
Subsequently, a broad stress test has to check the three memories connected to the SYS-MEM arbiter. That test writes a high data volume into each memory and checks the functional connectivity of all addresses. The implementation section 5.3.1 explains how to verify the memories accurately in more detail.

XPP Array

The array processor architecture is a new arising technology. Even though the design has been extensively tested in simulations, small XPP Array tests are appropriate for the real semiconductor chip. In particular, the timing might be faulty under operational speed whereas the overall logical behaviour should not cause errors. Further, the XPP Array contains functional and path components. The first class has a good testability since each ALU can be programmed directly so that the tester has full control over what the array does. The latter might be problematic for a complete test since a routing algorithm automatically generates paths between Array-PAEs. A manual connection of Array-PAEs is not possible. Therefore, a test only verifies the timing by provoking stalling and no-stalling.

Between the FNC Container and the XPP Array is a clock-domain crossing that could be sensitive to crosstalk faults as the analysis has indicated. Thus, that crossing area must be stressed under operational speed during commissioning, showing its reliability. The verification can be achieved by transferring data between both sides using as many ports as possible.
Further, the XPP Array includes an own memory component that is accessible by the 16 RAM-PAE objects. After proving that the timing of the XPP Array and the clock-domain crossing is as expected, the internal memory can be stressed with the same method as the one applied to the board memories.

XPP-Core

Two internal memories, various memory access components, and a network with direct connections surround the XPP Array. The various DMA components are a well-known technology in hardware architectures. Only the RAM-IO component is custom-built and provides 32-bit memory access for the 16-bit computing XPP Array [PAC09b]. However, those components are comprehensively verified and do not contain critical features such as a clock-domain crossing. Consequently, particular tests targeting only those modules may not be necessary. The X-FIFO and X-RAM memory tests utilise memory access components and interconnections to demonstrate the proper timing. Therefore, the X-RAM may be stressed with either the 4D-DMA or RAM-IO module, and the Linear DMAs are helpful for checking the X-FIFO module.

The remainder interface to test is the Stream-IO block that is interconnected with Channel Links. That module can be verified by sending data through the interface to the board memory and back. Even though the Stream-IO ports are suitable for bidirectional connections, the chosen board limits a complete test. In particular, the single-chip board and the BLTS only provide one Channel Link IN at port 0 and one Channel Link OUT at port 2. Port 1 and 3 are interconnected with a loopback and testable in both directions.

Full load

After stressing all critical internal features and interfaces separately, a comprehensive test, utilising as many parts of the HPDP as possible, has to show the stability of the power supply voltage. Adverse effects on signal integrity influencing the correctness must not occur. Thereby two strategies are possible. Either a test program employs as many as possible components during execution or one of the researched algorithms demonstrating the computational capability of the XPP-Core is used. For example, such algorithms are developed in other student works to investigate the portability of image processing algorithms on the array architecture [Tru15, Bar14]. However, there is an open question about the integration of the algorithms into the testbench developed throughout this work. The algorithms are implemented for a plain XPP-Core ignoring peripheral components like SpaceWire. Section 5.3.5 introduces the integration of the ported image detection algorithm for space debris detection [Tru15].

4.4.3 Summarizing

Table 4.3 summarizes the proposed tests. Compared with figure 2.1, all interfaces and internal modules are included in the commissioning concept.

Proposed tests	
XPP Core	Short timing tests → XPP Array, DMAs, X-Bars
Memories	Functional connectivity checks with partial March test → Board and internal memories, and FNC caches
Streaming interfaces	Connectivity and stress tests → SpaceWire, Stream-IO
Watchdog	Long-run functionality test
GPIO interface	Pin tests regarding board specification
Power supply	Stressing the whole HPDP chip to check the power supply voltage

Table 4.3: A short overview of the proposed tests

Additionally, the components SYSMEN Arbiter, X-Bar network, XPP-configuration, System controller, L1 I and D-Caches, and address decoders are also tested besides.

5 Implementation

The previous chapter enumerated required tests for the commissioning. This chapter deals with the realization by showing the implementation step by step and addresses rising issues. Each test will be applied to the first delivered HPDP chip and single-chip board. Since the practical demonstration is not part of this thesis, the implementations are thoroughly developed for and tested in a given ModelSim testbench.

5.1 Tools and target platform

The development environment does not provide real-time hardware and the complexity of the VHDL design leads to a low simulation speed in ModelSim. Therefore, the simulation tool xsim is employed. It provides a fast system-level simulation of the XPP-Core and is convenient for developing and debugging but the missing HPDP peripherals are limiting the application. Thus, the tests need the variability of running in three operational applications where each of them represents a different model of the processor design.

- xsim
- ModelSim
- Board Level Test System (BLTS)

The differentiation is realized by various Makefile-targets. Every target compiles the source code and runs the aimed environment with particular arguments.

5.1.1 xsim

The XPP-Core simulator xsim runs machine code based on Assembler or C. The same machine code is executable by the real HPDP chip. The two available FNCs are programmable via two separate main-functions. Implementing concurrency is, therefore, simple to establish. The system-level simulation is much faster than ModelSim so that comprehensive programs need an acceptable amount of time.
However, some HPDP parts such as SpaceWire interactions and the Watchdog registers cannot be addressed, and the source code must provide two separate compilations using a preprocessor target. Otherwise, xsim throws an error due to prohibited write or read

accesses, which may also lead to an unexpected behaviour of an algorithm executed on the XPP-Core.

The preferred programming language for coding hardware programs is FNC-PAE-Assembler because the compiled code is smaller than code compiled with the FNC-GCC. This different infect the startup overhead and is typically critical for device simulators [PAC09f]. Nevertheless, programming with C is more convenient, and the startup overhead does not matter for real hardware tests that are much faster as simulations.
Next to the FNC-GCC, an FNC-ELF-GCC compiler is also provided. The target format is the Executable and Linkable Format (ELF) because only this is interpretable of a real HPDP chip. In some cases, ModelSim runs into an unexpected behaviour while using FNC-ELF-compiled code. For that reason, the Makefile offers a flag to switch quickly between both demanded compilers. The evaluation section 6.2.3 presents that problem in more detail.

5.1.2 ModelSim

A very precise RTL model of the HPDP chip represents the VHDL design used in the ModelSim testbench. All HPDP components and almost the entire board is included, and the SpaceWire protocol can be simulated. The only missing parts are the board modules. Therefore, the exact Channel Link configuration to the Stream-IO ports and the connections to GPIO pins are not provided. However, both interfaces can be controlled and observed.
An SpW-script controls a ModelSim simulation and is based on a Tcl-script (Tool command language). The SpW-script steers the flow of each test program, and contains commands for ModelSim management, and sending and receiving SpaceWire commands. A disadvantage is the rigid sequential flow, meaning that loops and branches are only possible under some restrictions. In particular, advanced Tcl instructions and ModelSim management instructions are not mixable. Figure 5.1 depicts the interaction of an SpW-script and ModelSim. The latter starts with loading the HPDP VHDL-design and runs the given SpW-script sequentially. Additional files like the FNC-program may be loaded.

Compiler challenges

A given FNC-GCC compiler translates the FNC-program into machine code. Initially, the compiler is only adjusted for the XPP-Core, and not for the final HPDP design, which leads to some challenges in program development for the HPDP.
The original compiler assumes to have three 32 Mbyte memories connected to the SYS-

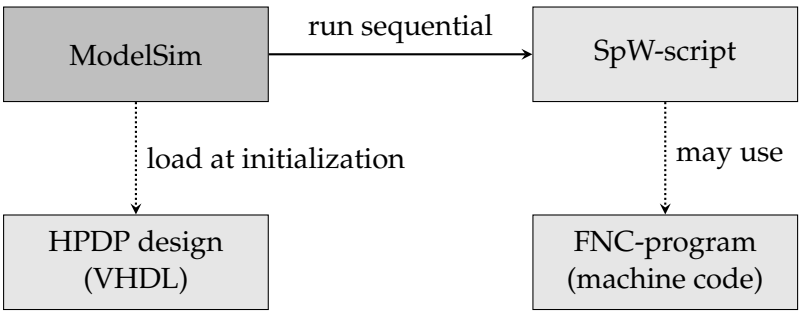


Figure 5.1: Schematic representation of the ModelSim testbench which simulates the HPDP design. A SpW-script contains SpaceWire instructions and advances Tcl-commands for monitoring ModelSim.

MEM arbiter and does not make a distinction between fast and slow memory. The final 512 Mbyte SDRAM connected to port 1 is much quicker and bigger than the 1 Mbyte EEPROM connected to port 0. Hence, the compiler should use the board memory connected to port 1 organizing stack and heap. A more verbose discussion is given in section 5.2.2.

Another concern is the Stream-IO block that is interconnected with Channel Links to other boards or instruments. The boot code, customized for xsim, set the four Stream-IO links in bypass mode because the xsim tool deals with files as an instrumental dummy. That configuration is not useful in ModelSim or on the real device. Consequently, an FNC-program has to reverse the boot settings and configure the Stream-IO ports correctly. Alternatively, the boot library could be modified, and the source code must provide an extra configuration during development with xsim.
Moreover, modules such as Watchdog and SpaceWire are missing in xsim. If the simulation tries to access a address of the missing components, errors occur, and the program behaviour is not as expected. Hence, preprocessor flags have to provide different versions depending on the target platform.

5.1.3 Single-chip board and the HPDP chip

ModelSim und xsim are just tools to simplify the development and verify the functional behaviour. The intended target platform is the HPDP chip placed on board. As ModelSim executes a complete HPDP design, the developed FNC-programs should not need further adjustments. On the contrary, the Tcl-scripts are implemented for a ModelSim environment that has much more functionality for monitoring and controlling than a real HPDP chip has. Moreover, the Board Level Test System and its UMDS Runtime System executes a C# script that is different from Tcl. Consequently, a SpaceWire command file must

be translated into the required format to be interpretable of the UMDS Runtime System. That task is excluded from this thesis, only a description of all containing commands is provided and shown in table 5.1. Figure 5.2 depicts the translation flow.

Figure 5.2: Each test case is steered by SpaceWire commands (and Tcl commands for monitoring ModelSim). That test sequence is scripted in a Tcl-script and translated by a Tcl-interpreter into an SpW-script, which can be interpreted by ModelSim (an example is given in appendix A.2). Dotted: Another translator may convert the SpW-script into a UMDS script, which is excluded in this thesis.

Script commands	
SPW_[1/2/3] *byte*	Sends the given *byte* via the SpaceWire link [1/2/3]
SPW_[1/2/3] cp *byte*	Waits until receiving a byte from the SpaceWire link [1/2/3] and compares it with an expected value; ModelSim exits if the values are not equal
#e# *arg*	Prints *arg* as a comment; may be interesting for observing the test flow
run *x* [ns/us/ms/s]	Continues the simulation for *x* time units and executes the next script instruction afterwards; perhaps ModelSim only
tcl> *arg*	Executes the given tcl command *arg*; ModelSim only, mostly used for monitoring the current simulation state

Table 5.1: Explains the most interesting SpW-script commands

The two most important commands are for SpaceWire control. The first one sends a given byte to the selected port of a SpaceWire interface; the second command arranges a waiting until a byte is received at the given port. The function also compares the received byte with the given expectation and stops the simulation in the case of a mismatch. The comment command is optional but could be useful for monitoring the test sequence also with the UMDS runtime environment. ModelSim requires the usage of the *run* command sometimes because the an action must wait until the result of the previous step has applied. In particular, in the boot test follows an FNC-halted check after booting for about 1 ms, which needs about 2 minutes of ModelSim computation. Therefore, ModelSim has to break the script execution until the FNC has finished. The last Tcl-command, tcl> *arg*, is ModelSim only and not required for the real target platform. It serves mainly monitoring purposes of the ModelSim state and is not applicable on real hardware.

Appendix A.2 shows the C-Code, implemented Tcl-script, and resulting SpW-Script of the Boot test as an example.

5.2 Test flow realization

Each test case is unique and needs an adapted test sequence. For example, the first test switches on the LEDs connected to GPIO via SpaceWire, and is much simpler and requires fewer steps than the Full Load stress test. Also, the result verification of the LED test is done by additional SpaceWire commands instead of an FNC-program. Figure 5.3 depicts the most complex test flow with an FNC-program and multiple response messages. Depending on the test case, a test script does not contain all listed steps. The basic-functional tests, except the boot test, does not have an FNC-program, and many test cases only have one SpaceWire response. The following subsections give a detailed explanation of each step.

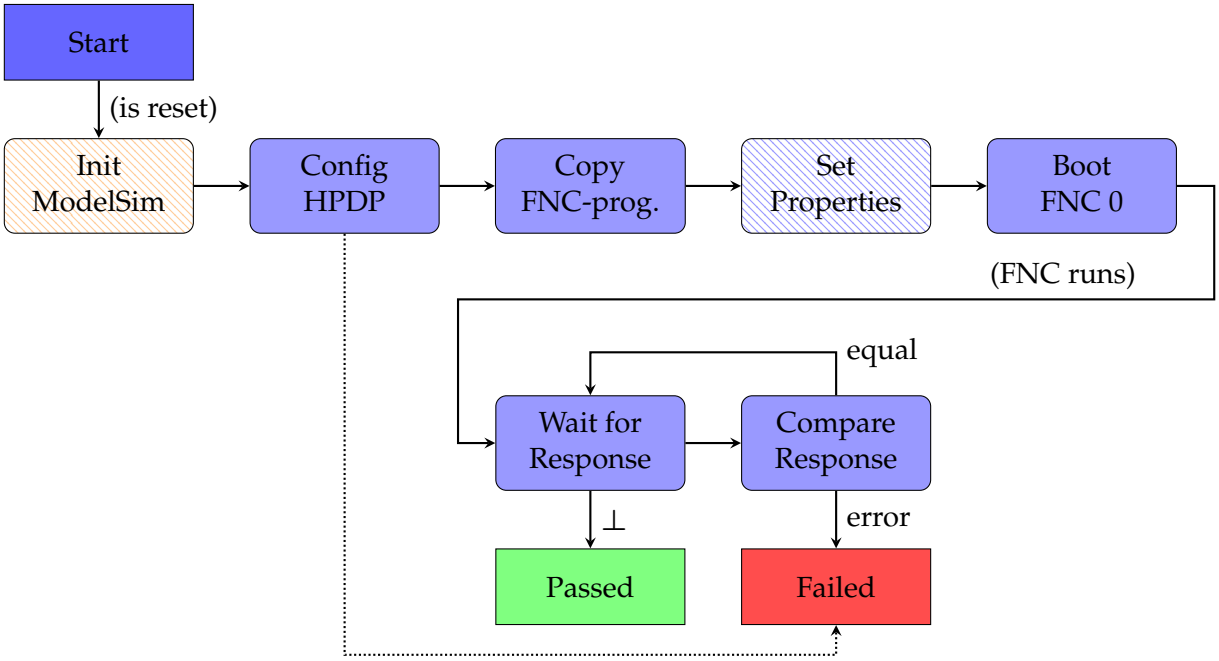


Figure 5.3: The flow chart shows the final implemented script sequence. The hashed nodes are new compared to the conceptualized strategy illustrated in figure 4.5 (the orange node is required for ModelSim tests, and the blue one extends the adjustable script technique).

5.2.1 Configuration phase

As the analysis has shown, a test should start after a chip reset to avoid side-effects between consecutive test executions. A reset sets all registers in its default state, but not all default values are appropriate for a test case. Therefore, every program begins with a configuration step, which is similar for all test cases. A common Tcl-file provides the setup procedure allowing an easier maintainability of the test suite. For instance, each program calls the same configuration process to ensure that there is only one code section that needs to be adjusted if the memory layout changes. It is also easier to use two different setups, one for the given ModelSim testbench and one for the real single-chip board. The next section explains why two different memory layouts are necessary.
The start process performs several sub-steps. First, the appropriate SpaceWire, memory, and Stream IO clocks are set, and the bypass PLLs is activated. Second, the script configures the SYSMEM arbiter and removes the memory write protection. Last, the procedure needs to wait until a legal PLL status is set.

5.2.2 Program copy

Before the FNC0 can start the boot process and execute the chosen test program, the machine code needs to be copied into a memory of the HPDP chip. That may be the non-volatile EEPROM at memory port 0 or the volatile SDRAM / SRAM at port 1 and 2.

Copy the machine code

After the configuration phase, an FNC-program is copied into the chosen memory via the SpaceWire interface. Thereby, the machine code is given in binary format after compilation, and the tester needs to disassemble the machine code and transfer it in the right order to the HPDP memory. The command

xfncasm -v *binary sysmem.in*

does the conversion of the binary. Afterwards, a simple script can extract the content line-by-line and send it by SpaceWire packets. The *xfncasm* outcome contains 64-bit lines in Little-Endian order. The script must reverse every line to Big-Endian, and separate it into two 32-bit SpaceWire packets. Figure 5.4 illustrates the process.

FNC-program location

As introduced in section 4.2, the single-chip board and the HPDP chip provide three different internal and external memories which are capable of storing instruction code - one

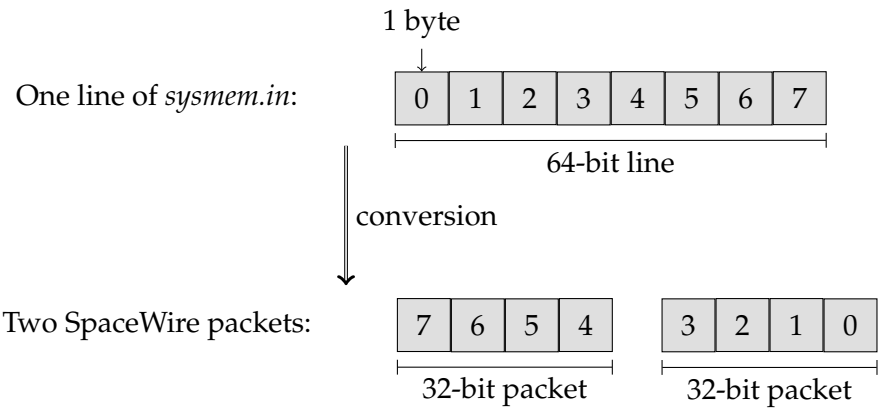


Figure 5.4: Every line of a program code given in file *sysmem.in* needs to be reversed and split into two SpaceWire packets.

8-bit EEPROM, one 64-bit SDRAM, and one 64-bit SRAM. The SDRAM with a size of 512 Mbyte is much bigger than the other two memories. However, the 1 Mbyte EEPROM may be sufficient for the simple test programs. There are two major disadvantages to use the EEPROM for exhaustive testing. First, it is much slower than an SRAM or SDRAM, and second, many write cycles reduce the lifespan of the EEPROM much faster. Hence, choosing the fast SDRAM for multiple tests is the best option.
The HPDP chip is in non-volatile mode after a reset. That means the processor boots from address 0x0. Usually, the EEPROM is located at that address, and the SYSMEM Memory arbiter must be configured to route the address 0x0 to memory port 1 where the targeted SDRAM is located.

ModelSim adjustment

A couple of obstacles have shown up during the implementation. The code copy procedure in ModelSim is accelerated by using the *mem load* command that copies the instruction code directly into the memory without using SpaceWire. That takes only a few milliseconds instead of multiple minutes (real time). Unfortunately, an accurate layout model of the memory must be provided. Otherwise, the command cannot be used. That situation occurs when using a model of the preferred SDRAM. Thus, only a SRAM module is usable during development because waiting minutes just for copying the code is not convenient.
Moreover, another problem has arisen during the implementation process. The synthesized SRAM module for memory port 1 of the given testbench has a faulty timing that makes a correct behaviour impossible during the boot process.
As a consequence, the FNC-program is copied to memory port 0 where a fast 8-bit SRAM

is located for ModelSim development. The HPDP configuration is adjusted and provides two different SYSMEM arbiter setups. One is for ModelSim with an SRAM at port 0, and the other one is for the real chip with an EEPROM at port 0 and boot address at port 1. A different command line argument for the test scripts guarantees the right behaviour for each situation. However, the configuration case for the real HPDP cannot be tested.

5.2.3 Keep the duration optional

The idea behind stress testing is using a component or the entire processor under operational speed for a long duration. A discussion about the optimal duration is not done in this work since the XPP Array architecture, and the 65-nm rad-hard semiconductor material is brand new and needs further investigations. Consequently, functional stress tests have the possibility to run a certain number of repetitions or endless. The test user can break a test at every time without damaging the chip or board. The goal of a long duration test is finding sporadic hardware faults those only appears under certain conditions.
Another goal is clean programming, avoiding subsequent changes in C-code by the test user. All adjustable variables should be available in the corresponding script so that the user only needs to change those in the case of testing a module with different properties or number of repetitions.
A test script has two possibilities of transferring the selected properties to the HPDP chip. First, writing the values into free board memory before booting, or second and less complicated, writing the values into the Scratchpad Registers via SpaceWire. Two of those registers are provided and offer 32-bit storage altogether. Using the both Scratchpad Registers is less complicated and free of side-effects. In turn, 32 bits are not much if many settings with high numbers are required.
Using board memory as a property section does also have a drawback. Side-effects may happen if the properties overlap with code memory, heap, or stack. In particular, the FNC could override the given properties during the boot process and before reading it. A proper solution is using the third memory module, whereas the first and second memories are reserved for code, heap, and stack. Both strategies are implemented depending on the property size of each test case.

5.2.4 Waiting for response

Each functional test has one or more results which need an evaluation at the end, verifying whether a test passed or failed. There are two main approaches to achieve that goal. Either all result values are sent to the tester or an FNC checks directly the outcome on-chip and only sends a few status bytes to the tester.

If the latter approach is employed, the FNC-program needs the knowledge about the expected outcome and the assumption that a processing unit compares accurately must be true. According to the analysis made in chapter 3, that assumption is given. Otherwise, far more basic instruction must be checked before testing the integration of modules.
The first described comparison approach has a disadvantage as many of data may be transmitted over SpaceWire. In particular, a complete X-RAM test has to send at least 4096 32-bit packets. Equal to the code copy problem, that would influence the duration of ModelSim executions a lot. On the contrary, the FNC already knows the expected test outcome because the test patterns are given in the memory or created by a generator running on the FNC. Random patterns are avoided by using reasonable test values. Thus, an overhead of sending reference values does not occur, and the on-chip verification remains the preferred solution.

The final response is different for every test depending on the tested feature. Some programs return a couple of bytes just at the end. Other do it after every repetition. A SpaceWire packet has a length of four bytes, and due to a ModelSim restriction, the last byte of a response packet is always the crucial pass-or-fail-byte. That means that the given ModelSim testbench breaks immediately if a received byte is unequal to the expectation. As a result, the first three bytes of a SpaceWire response packet can be used for general information such as details about the faulty module or the repetition. That compresses the reply message because of fewer SpaceWire packets. More detailed information can help to spot a possible defect. In particular, if the repetition number and pattern type differ between several faulty test executions, the defect may be classified as a sporadic hardware fault instead of a wrong logical behaviour.

Using the SpaceWire DMA module

The SpaceWire DMA module has six primary registers for a data transfer that contains a header and a data field. The start and end addresses of the header and data section need to be set before starting the data transfer. That mean that the FNC stores the content of the header and the data field in a free memory location, and writes the border addresses into the four SpaceWire registers. Both data fields should be declared as non-cacheable. Otherwise, the content can still be in the L1 D-Cache when the DMA transfer starts, because the SpaceWire does not initiate an FNC-cache cleaning before starting a transfer. Cacheable variables could lead to an unexpected behaviour since the SpaceWire module would transfer data with unknown content. The header section contains routing data at the beginning and may include additional information such as the size of the data field or the task id to distinguish multiple responses. The implemented header uses a protocol to

differentiate between parallel tests on multiple chips. Section 5.4 discusses the protocol in more detail. Unlike the header, the data field may be empty.
The data delivery starts when the second bit of the DMA control register is set. If the SpaceWire module has finished the transfer, the corresponding interrupt is set. Therefore, busy waiting is possible while an FNC routine waits for the completion of the current transfer. The FNC-program has to delete the interrupt if a subsequent DMA transfer should also be observable.

The given ModelSim testbench forces some particular implementations which are not necessary for the real processor check. One example is putting the pass-or-fail-byte at the end of a 32-bit response packet since the test execution breaks immediately if an unexpected byte occurs. Another particular adjustment demands the time-out after 100000 cycles if nothing happens at the SpaceWire ports. That is a convenient behaviour for the previous verification which tests functions using SpaceWire commands only. In turn, the time-out is not useful for commissioning tests where an individual FNC-program may perform much longer than 100000 cycles. Two strategies can be applied to avoid a time-out. First, the *RUN* command is used to stop the cycle counter, or second the time-out logic is deactivated by a force command. The latter approach is adopted because it is much easier to use than determine the expected runtime of a test program.

5.3 Test case implementation

This section describes particular implementations of various test programs in more detail.

5.3.1 Memory connectivity testing

A variable software strategy is implemented in addition to the comprehensive BIST structure. A free usage of different parameters allows the creation of various test pattern to perform more memory tests from the operational side.
Concept section 4.3.4 has introduced the length 9N algorithm for memory testing. The march test applies different patterns to every address of a memory-under-test [TNF01]. Since that should already be completed by BIST, the commissioning tests mainly verify the functional connectivity between a memory module and the processing units. That can be achieved by using the counter pattern in both directions. Figure 5.5 illustrates why writing in both directions is necessary. In the worst case, a multiple-write fault may be overlooked if the test runs only in one direction. A more comprehensive solution would be reading the entire memory after every write command, which would cause a much higher complexity and is, therefore, not implemented.

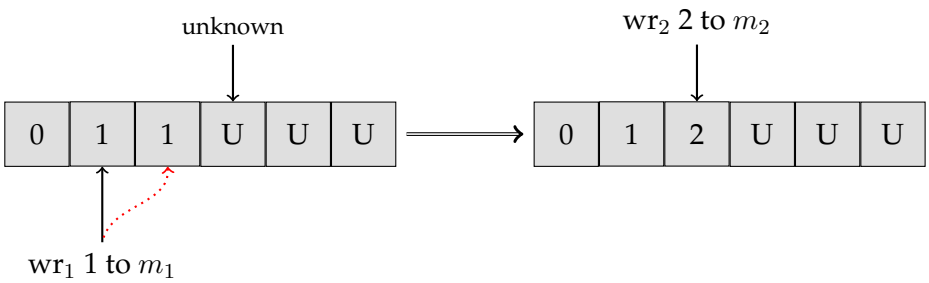


Figure 5.5: If the counter pattern for memory testing is only used in one direction, a multiple write access fault may be overlooked. Here, the write command wr_1 accesses also the right neighbour cell which is going to be overwritten by the next write wr_2. Thus, the read-verification cannot detect the fault.

Nevertheless, stressing the memories with also other patterns may be attractive. Hence, each memory test script contains a start address, end address, 32-bit pattern (= start value), increment, end value, direction, write cycles, and read cycles. The number of read repetitions r is performed after every write cycle w, meaning that a test executes $r \times w$ read cycles in total. This approach should provide the option to avoid too much stress to the EEPROM module. Pattern such as 0b0000, 0b0101, 0b1010, 0b1111 in both directions or with different increments are possible. Also, the duration is entirely controllable.

Concurrency

Test cases like the FNC-TCM test using both FNC-PAEs, and have, therefore, a concurrent program flow. Both processor units can communicate either via the FNC-Crossbar and transfer events or data among each other, or utilise the board memory as shared memory. A shared L2 data cache is not available. Atomic access to the board memory is possible, although there are not needed for the test cases. [PAC09g]

Whenever a synchronization between both FNCs is required, the board memory is used, because the C-implementation is less complicated than configure and transfer via the FNC-Crossbar. Shared variables need to be declared as `volatile` to prevent the compiler from storing those variables into L1-cache of an FNC. There is no runtime logic available which checks whether a utilised variable is still in the cache of the other FNC or not. Consequently, the runtime behaviour is undefined for a shared variable which is not `volatile`.

Further, an FNC also works in parallel with the XPP-Array. If an FNC generates test data that is accessed by DMAs or the SpaceWire afterwards, the data must also be `volatile` to ensure the expected behaviour.

5.3.2 XPP Array

The implementation of the XPP Array test is kept small since no exhaustive tests are required according to the analysis. The add operation with carry (`addc`) is programmed at one ALU-PAE. The input paths are provided in different configurations to test different timings of the input parameters. Further, four different operands and two events are used in all possible combinations. The event represents the carry bit. The XPP Array configuration sends the result back to the FNC, which does the same operation with the same operands and compares both.

5.3.3 Interfaces

The HPDP chip has two high-speed interfaces for purposes of controlling and data transfer. Both, the SpaceWire and the Stream-IO interface, are emulated in ModelSim. Bytes are sent via SpaceWire as introduced in table 5.1. The Stream-IO communication is imitated by files, implying that one input file and one output file for each port must be available or is generated during execution. Testing a loopback, as it is available between port 1 and 3 on the single-chip board, is not possible because ModelSim cannot access the same file for one input and one output port. A loopback must directly be available in VHDL.
The script for the interface verification provides an optional repetition value to stress the interface as long as needed. Also, different clocks can be chosen, and the test data is easy expandable by the Tcl-script.

5.3.4 Watchdog timer

The watchdog module has to be tested during commissioning since long-run simulations are costly in ModelSim. The implementation provides two options. On the one hand, a stand-alone test script, motivated by the given testbench, can check the watchdog functionality for any arbitrary timer value. On the other, an FNC implementation sets a given timer and checks whether the interrupt is set too early or too late. However, with an FNC-program is the border not as accurately testable as with a script. In particular, the difficulty is to wait exactly for the required amount of cycles after the program resets the timer, without analysing the FNC instructions.
One solution would be using the assembler instruction `NOP`, letting the FNC doing nothing for one cycle. However, bypassing a long duration requires many NOP instructions, which, in turn, makes a loop preferable. That needs further investigations of timing because branching also takes FNC cycles. Following the reasoning, using the FNC solution is possible, but the test script solution using only SpaceWire commands for verifying the watchdog is recommended.

5.3.5 Stressing the entire HPDP chip

An on-chip test program should stress the entire chip showing the reliability of the power supply voltage. Two possibilities are discussed in section 4.4.2. The first test uses as many as possible components at once, and the second shows the integration of a researched algorithm for the XPP-Core into the commissioning testbench.

The full load test

The aim of the Full Load realization is to utilise as many as possible components of the HPDP by avoiding a too complex implementation. Therefore, all components are allocated to two tasks, which are executed on both FNCs in parallel.
After booting and setting up the configuration by FNC 0, both FNCs start with the task execution. The FNC 0 has two sub-tasks. One steers a data transfer from board memory to X-RAM over the XPP Array, and the other sends the current progress status via SpaceWire to the tester. The other processing unit, FNC 1, controls two sub-tasks, as well. A data transfer between Stream-IO ports and memory is performed, as well as a transfer between the X-FIFO module and memory. Both sub-tasks utilise the three Linear DMA components. Altogether, both FNCs, streaming interfaces, all internal and external memories, as well as memory access components are used to stress the whole HPDP. Only the XPP Array is not completely utilised.

Integration of an algorithm

All commissioning tests shall demonstrate the operational capability of a delivered HPDP chip. In the case of positive feedback, the chip can be employed to evaluate researched algorithms in praxis. One of those works represents the implementation of an algorithm for on-board space debris detection. The algorithm is realized for an XPP-Core using xsim as a simulation tool. The final HPDP design and a SpaceWire interface are not included. The input image is provided in a file and loaded via the Stream-IO interface, whereas the output image stays in memory. [Tru15]
Consequently, some adjustments are required to make the source code ready for an operation on an HPDP chip. The following enumeration lists general required steps to integrate an algorithm into the commissioning testbench.

- Replace the Makefile with one of the testbench and set the name of the application and Stream-IO files if required
- Add a Tcl-script to automatise the configuration step

- Add Stream-IO and SYSMEM arbiter configuration to C-code (done by a library call)
- Add a SpaceWire response if needed
- For ModelSim, replace calls of the XBAR-Connect C-API by assembler

The first bullet point is only necessary if the source code is used for different platforms, avoiding the creation of an own Makefile. The second and third point offer the possibility to include created Tcl and C-libraries. Those provide useful functions to configure the HPDP chip probably. Also, the Tcl-library manages the code copy procedure via SpaceWire. If a SpaceWire response should be added, the C-library offers tested functions. The last bullet point addresses an arisen problem. The complex X-Bar network can set up with assembler instruction, which is not very handy. Therefore, a comprehensive C-API provides concise functions to connect X-Bar ports. xsim and the C-API work together without issues. However, ModelSim runs into memory errors and an unexpected behaviour calling functions of the C-API. The problem is similar to the arisen FNC-ELF-GCC issue, which the evaluation section 6.2.3 explains it more detailed. A workaround is to use another tool which generates fewer X-Bar assembler instructions. However, the C-Code must be rewritten into another format.

Altogether, the enumerated steps are applied to the given code base of the space debris detection algorithm. The first three steps take code from existing commissioning tests. Whereas, the last two need costly code changes.

5.4 Multi-chip board (HPPM) testing

The focus throughout this work is on commissioning testing of the first HPDP chip operating on a single-chip board. Nevertheless, a multi-chip board, which has space for up to four chips, is developed in parallel. Differences between both boards are the SpaceWire and the Stream-IO interfaces. As the section 4.2 introduced, the four chips on a multi-chip board can transfer data to each other over direct links. There are no shared memories or other concurrency critical components on a multi-chip board.
Consequently, all single-chip test programs can also be used for the multi-chip board. Two additional tests may be useful then. One which checks the direct communication ability across the board via SpaceWire, and one for the direct Stream-IO connections. By the way, those tests are not testable in ModelSim because there is no multi-chip environment provided.

However, there is one design problem using one tester and multiple chips at the same

time. All chips use the same SpaceWire to receive data from the tester and send a response back. Accordingly, the test environment needs two adjustments. On the one hand, the Tcl-script needs to set right header values for the SpaceWire protocol sending the data to the right chip. On the other, every response message must have something like a unique process id and probably also a message id. Otherwise, the tester cannot identify the sender of the message.
Moreover, the design of the multi-chip board allows the interconnection of several multi-chip boards via Channel Link connected to Stream-IO ports. If a multi HPPM program should be developed, this link needs to be activated by activating GPIO pin 0 in output direction.

6 Evaluation

A real HPDP chip is not provided, and the introduced test setup cannot be utilised. The first part of this chapter describes the verification methods, which are used to check the correctness of the implemented test programs. Subsequently, the runtimes of ModelSim and xsim simulations are presented. The chapter completes with a discussion of arisen problems.

6.1 Verification

A realizable verification strategy is required to ensure the correct behaviour of each test program. In other words, further tests have to show that every test program finds the possible, aimed faults and provides a suitable error message. Three various methods are applicable. First, SpaceWire commands can manipulate memory cells or register values at runtime. Second, the C-code may provide a routine which changes the state during execution, or third, the ModelSim ability to force a command to a faulty value may be applied.

6.1.1 Inserting faults via SpaceWire

SpaceWire commands can induce errors by manipulating memory cells or registers during a test execution. Here, the steering Tcl-script provides logic that sends SpaceWire commands after the first iteration of the test to manipulate the investigated component. If the next repetition responses with an error message, the fault was recognized. Otherwise, the realization of the test program is not sufficient.
That method, however, has several drawbacks which make a correct implementation of the verification tests difficult. One limitation is that not all registers or memories are addressable via the SpaceWire, i.e. memories such as X-RAM or TCM are not manageable from SpaceWire. Another issue concerns the right timing. The execution runs for an uncertain amount of time between monitoring the state of the HPDP and sending manipulation packets. That makes an injection of a timing-critical error impractical. Additionally, manipulated register contents may lead to an unpredictable behaviour, so that the HPDP cannot send a response. Then, it is not clear if the missing response is caused by the manipulation message or something else.

6.1.2 Using faulty C-routines

The timing-critical disadvantage of the previous method can be eliminated by directly handling the faulty routines in the C-code. In particular, while the FNC 0 runs, the FNC 1 could execute a function that manipulates cells employed by FNC 0. That approach also has some restrictions. The manipulation is only as powerful as the regular test code since the instruction set is limited and X-RAM or caches are not directly manageable. Further, the first four basic-functional tests do not have an FNC-program to execute such a routine. Another significant disadvantage concerns the extra pieces of C-code, which may reduce readability and extensibility if the program needs further adjustments later.

6.1.3 Forcing signals motivated by Mutation testing

ModelSim is already widely used throughout the implementation process and provides a command to force any signal of the HPDP chip to a particular value. That method is related to mutation testing.

Mutation Testing is a widely-research fault-based testing technique, where the original program is seeded with simple syntactic changes to create a faulty version, called mutant. A set of mutants is executed on the input test set to assess the quality of the test set. If the result of a tested mutant differs from the result of the original version for any test case of the input test set, the seeded fault is detected. A mutation score can be calculated and indicates the quality of the input test set. The Mutation technique is mainly used in software testing. Verification of hardware designs is also possible by either seeding a fault into the VHDL/ Verilog code or by forcing signals to wrong values. [JH11, SBR07]

Changing the HPDP's source code requires knowledge of the used programming language and extensive architecture. Usually, many mutants are necessary to reach a high mutation score.

Forcing a signal to a wrong value has the same timing-critical obstacle as the SpaceWire approach has. In particular, if the signal force comes too late, the FNC-program has already finished the critical part, and the fault is perhaps not detected. Otherwise, if the signal is changed too early, the force could affect a previous process and lead to another unexpected failure. ModelSim provides some options to observe the state of the HPDP and pushes a signal at the right moment.

6.1.4 Final realisation

Altogether, using the force command of ModelSim would be the best solution. However, the HPDP layout comprises an elaborate design with thousands of signals. Knowing the right signal to force requires a broad knowledge base of the HPDP implementation. Sufficient documentation is not available. For example, the addressable registers of the periphery are explained in documentations, but the names and addresses do not coincide with the signal names visible in ModelSim. Thus, finding the right signals is costly. Small, implemented C-Code routines or sending manipulation messages via SpaceWire is mainly used to verify the capability of error detection.

6.2 Results

6.2.1 Runtimes

Table 6.1 presents the runtime of each test executed in ModelSim and xsim. The latter measures the simulation time in cycles without some periphery modules like the SpaceWire. The first four tests shall show the general responsiveness and do not have an FNC-program that could run in xsim. The watchdog test is only executed by Tcl-commands since xsim does not contain the watchdog module and ModelSim is too slow for a long-run test. The runtimes of the ModelSim simulations include the whole script flow from the HPDP configuration until the response checking. ModelSim measures the simulation time itself. One millisecond simulated in ModelSim needs around two real minutes. The code copy procedure from the tester to the HPDP is only demonstrated once in ModelSim and takes around 51 ms. Each listed test result in table 6.1 is simulated with the *mem load* command that skips the copy procedure.

On the one hand, ModelSim contains more modules and calculates the entire SpaceWire protocol. On the other, xsim simulates some additional debug functions like the print commands for observing the current state in the terminal. Consequently, both measured durations does not correlate. In particular, the Boards ID test only requires slightly more time than the reply test in ModelSim but needs about 50 percent more cycles in xsim. Another big gap is observable between the RAM-PAE and X-RAM test. The reason could be the extensive use of the print statement in the former one (the code for printing a string in the terminal is ignored by the preprocessor if the target platform is ModelSim because the stream cannot access to the ModelSim terminal).

Runtime of each test programs			
Test name	Conditions	xsim (in cycle)	ModelSim (in ms)
GPIO		-	0.158
MEM		-	0.947
SPW		-	1.208
STRESS	1024 Mbyte each MEM	-	5.141
BOOT		9817	1.379
REPLY		24423	1.501
BOARD ID		38447	1.571
WATCHDOG	script only. timer=1500	-	0.652
FNC-XBAR		48850	2.311
FNC-TCM	complete. 2 repetitions	3381920	6.987
FNC-MEM	990 hexaddr., 2w, 1r	335252	12.900
XPP-ARRAY		29710	2.242
CDC-FNC-ARY	30 repetitions	241698	3.506
	300 repetitions	2537772	11.671
RAM-PAE	complete, 1 rep	2157864	57.301
X-RAM 4DDMA	complete, 1 rep	1367487	60.413
X-FIFO	complete	756685	10.780
STREAM-IO	13 values	68359	2.643
FULL LOAD	4 rep., 64 Mbyte XRAM, 32 Mbyte XFIFO	129985	8.879
	8 rep., 256 Mbyte XRAM, 1024 Mbyte XFIFO	364027	23.351
Integration of [Tru15]	image 128 x 128	3144286	-

Table 6.1: The runtime in xsim and ModelSim of each test case under given conditions. The conditions are adjustable in the script to expand the runtime.

6.2.2 Integration of further algorithms

The given source code of the algorithm for space debris detection is integrated into the testbench. Some adjustments for SpaceWire support and memory usage were necessary to compile and start the algorithm in the testbench. The algorithm is calculation-intensive and runs around one real second for a 2048x2048 image [Tru15]. Therefore, constants and the size of the input image are changed to only process a 128x128 input image. That reduces the runtime significantly. Despite those adjustments, the full algorithm is only executable in xsim. ModelSim struggles while reading the input image at the Stream-IO port. The reason for this open issue could be the format or size.

6.2.3 Arisen problems

The implementation chapter has already described some challenges which been arisen during the development phase. Section 5.2.2 talks about selecting suitable memory modules for the three SYSMEM arbiter ports. Those should match the real test setup, which is not possible in ModelSim.

Using mem load

The test procedure converts the machine code into a format that fits the *mem load* command to speed up the initialization procedure. Tests have turned out that the assigned memory port throws an interrupt and a single bit error during the boot process of the FNC 0. Using the regular procedure and copying the code via SpaceWire instead of using *mem load* does not lead to a single-bit error. As the result, the acceleration procedure causes a small failure but does not influence the behaviour of the written test programs.

Difference in timing

One testing objective is verifying the correct timing of FIFO queues. Those are integrated into many parts of the XPP-Core, i.e, the X-Bars contains FIFO queues, as well as several XPP Array elements. Correct timing, therefore, must also work under stalling conditions, meaning that many packages are sent to a port before the first packages are consumed. Different results have appeared during the evaluation. In particular, the X-FIFO test case simulated in xsim can store up to 1041 packets before a deadlock occurs. In turn, ModelSim already runs into a deadlock after 1038 packages. Thus, the SystemC design stores four packages more than the Verilog/VHDL design. That is surprising because the analysis section discussed about different methods which check the similarity between both.
The reason could be located in the memory model since xsim emulates different memories than ModelSim does. Nevertheless, the exact reason remains unclear, and it is tricky to implement tests which provoke maximal stalling. Test user needs to adjust the values in the Tcl-script to test maximal stalling or solve an occurred deadlock.

FNC-ELF-GCC problem

The testbench provides a compilation process with the standard FNC-GCC, as well as with the FNC-ELF-GCC for micro controllers. The latter is the aspired format for the real HPDP. In fact, both compilers produce machine code interpretable by xsim or ModelSim. However, a runtime failure occurs while using the Executable and Linkable Format (ELF) and X-Bars for ModelSim. The error message "*Illegal Value of Address Bus during Read Cycle.*

Memory and Output Corrupted" reveals that the simulation tries to read from a uninitialized memory cell. Consequently, a simple boot test works, but a complex test using the network of the XPP-Core leads to an unpredictable behaviour, usually sticking in the current state. As a result, all tests running in ModelSim use the normal GCC format while xsim tests also work with the machine code compiled by the GCC-ELF.

7 Conclusion

7.1 Conclusion

The HPDP design is based on an XPP-Core that comprises a new array architecture. The current verification state of the HPDP architecture was analysed and open issues discussed. Afterwards, a commissioning concept was established and implemented. The performance of the on-board test software is shown in ModelSim since no HPDP demonstrator chip was provided to employ the created tests on real hardware. The test programs are bundled to a testbench. Universal Makefiles and shared libraries (one for C and one for Tcl) avoid redundant code and allow an easy extensibility. The order of the test programs is arbitrary, although a sequence of tests was recommended according to an incremental integration strategy.

7.1.1 Extensibility

The potential of the XPP Array architecture has been analysed by different case studies and previous student works. Those are developed using xsim, and their implementations do not deal with the full extent of a real HPDP chip. This study analysed and established a process flow from code copying, over booting, until receiving a response message, so that the source codes of other works can make use of the developed framework and run on a complete HPDP chip. The necessary integration steps for an algorithm of space debris detection are shown.

7.1.2 Using ModelSim

The array architecture requires graph-based algorithms. Therefore, a known algorithm of the streaming-based field needs to be reimplemented to take advantage of the array's potential. The analysis explains several drawbacks in developing a complete program with xsim for the HPDP. Also, the implementation has revealed significant issues of ModelSim. Testing complex algorithms with a large amount of input and output data is not comfortable for the developer. Debugging with ModelSim is only possible with a deep knowledge of the VHDL design, and mapping of source code and machine code running in ModelSim is hard to manage when a wrong behaviour needs to be investigated. Hence,

another developing tool is required to support the implementation and testing of a whole HPDP program in a sufficient amount of time.

7.2 Open questions

The most important open question is the operation of each test program on a real single-chip board and HPDP chip, also because of several issues with the ModelSim simulation. The memory setup of the single-chip board could not be used. The implementation chapter shows problems with setting up the memories equal to the planned board. Further, the emulation of the hardware connected to GPIO pins is missing, and the loopback between Stream-IO port 1 and 3 is not available in ModelSim.
The evaluation revealed that the machine code compiled with FNC-ELF-GCC leads to accesses of uninitialized memories and consequently unpredictable behaviour in ModelSim. Even though that problem does not occur in xsim, the right execution on a real HPDP is still questionable.
Finally, the SpW-script and Stream-IO input files require a further translation to be interpretable by the UMDS runtime system.

A Appendix

A.1 Technical data

Mem. port	Placement	Size	Type	ModelSim type
0	board	1 Mbyte	8-bit EEPROM	8-bit SRAM
1	board	512 Mbyte	64-bit SDRAM	32-bit SRAM
2	intern	4 Mbyte	64-bit SRAM	64-bit SRAM

Table A.1: The SYSMEM arbiter has three memory port. The first two ports are connected with a board memory, and the last one is linked to internal SRAM. The ModelSim testbench utilises different types with unlimited size.

XPP-Core memory	Number	Size	Direct addressable
X-RAM	1	8196 x 16 bit	yes
X-FIFO	1	1024 x 16 bit	yes
RAM-PAE	16	512 x 16 bit	yes
L1 D-Cache (Cache mode)	1 per FNC	8192 Bytes	no
L1 D-Cache (TCM mode)	1 per FNC	4096 Bytes	yes
L1 I-Cache	1 per FNC	32768 Bytes	no

Table A.2: All internal memories of the XPP-Core. X-RAM, X-FIFO, and RAM-PAEs are volatile memories supporting the XPP Array; the D-Cache has an optional TCM (Tightly Coupled Memory) mode making one half direct addressable by FNC instructions. [PAC09g]

Stream-IO port	Configuration
0	Input
1	Loopback to port 3
2	Output
3	Loopback to port 1

Table A.3: Stream-IO configuration of the single-chip board (multi-chip board: all four chips has a different configuration that does not matter here)

GPIO pin	Direction	Allocation
0	out	SpaceWire select
1	out	Channel Link select
2	out	LED 1 & Board-ID MUX input
3	out	LED 2 & Board-ID MUX input
4	out	LED 3 & Board-ID MUX input
5	in	Board-ID MUX output
6 to 15	-	unused

Table A.4: Allocation of the GPIO (General Purpose Input/Output) pins

Module	Planned frequency (in MHz)
System clock (XPP-Core)	250
(FNC-Container)	125
Basic clock of periphery	50
SpaceWire	200
Memory ports	100
Stream-IO	100

Table A.5: The planned frequencies for each part of the HPDP chip. The FNC-Container has always the System clock divided by 2. The SpaceWire, Memory port, and Stream-IO module are based on the Basic Clock combined with a multiplier and divider. [Bau15]

A.2 Code example

```
1  #include "hpdp_regs.h" // contains register definitions
2
3  int main(void) // instructions executed on FNC 0
4  {
5          // set direction to output
6          *GPIO_DIRECTION = 0x0000;
7
8          // activate output of pin [4:2] (LED on)
9          *GPIO_OUTPUT = 0x001C;
10
11         return 0;
12 }
13
14 int main_fnc1(void);  // instructions executed on FNC 1
```

Listing A.1: C-code of the test case *Boot test*

```
#!/usr/bin/tclsh
# load script with global functiona
source "$env(HPDPTEST_COMMON_DIR)/tcl/common.tcl"

setDefaultHPDPConfiguration

loadFNCcode

startFNC0

e # Verify that the FNC0 is running
rmap_rd SPW_1 FNC_IOREGS_FNCCTRL_START 0x0001

e # Run 700 us to bypass the boot time
waitForFNC 700

e # Verify that all FNCs halted
rmap_rd SPW_1 FNC_IOREGS_FNCCTRL_START 0x0000
e # Verify GPIO settings
rmap_rd SPW_1 GPIO_DIRECTION 0x0000
rmap_rd SPW_1 GPIO_OUTPUT 0xFFFF

reachedEndOfTest
```

Listing A.2: Tcl-script of the test case *Boot test*

```
tcl> coverage attribute -test BOOT_LED_TEST
[[Initialization]]
#e# # Load FNC Code (Fast - ModelSim only)
tcl> mem load -i mti.mem /hpdp_tb/g0/sram_8bit_memories__0/memport0_8bit_sram/memory/ram
#e# # Start FNC 0
#e# Write to [SPW_1] [FNC_IOREGS_FNCCTRL_START] 0x0001
SPW_1 30
SPW_1 01
SPW_1 60
SPW_1 93
SPW_1 20
SPW_1 00
SPW_1 13
SPW_1 00
SPW_1 30
SPW_1 00
SPW_1 01
SPW_1 00
SPW_1 00
SPW_1 00
```

```
21 SPW_1 04
22 SPW_1 76 # Header CRC
23 SPW_1 00 # Data Byte 3
24 SPW_1 00 # Data Byte 2
25 SPW_1 00 # Data Byte 1
26 SPW_1 01 # Data Byte 0
27 SPW_1 91 # Data CRC
28 SPW_1 EOP
29 #e# # Verify that the FNC0 is running
30 #e# Read from [SPW_1] [FNC_IOREGS_FNCCTRL_START] expect 0x0001
31 [[Header bytes]]
32 SPW_1 01 # Expected Data Byte 0
33 SPW_1 00 # Expected Data Byte 1
34 SPW_1 00 # Expected Data Byte 2
35 SPW_1 00 # Expected Data Byte 3
36 SPW_1 04
37 SPW_1 6F # Header CRC
38 SPW_1 EOP
39 #e# Wait for Reply Packet
40 [[Header bytes]]
41 SPW_1 CP 00 # Data Byte 3
42 SPW_1 CP 00 # Data Byte 2
43 SPW_1 CP 00 # Data Byte 1
44 SPW_1 CP 01 # Data Byte 0
45 SPW_1 CP 91 #  Data CRC
46 SPW_1 CP EOP
47 #e# # Run 700 us to bypass the boot time
48 RUN 700 us
49 [[Check final state]]
50 #e# # --- PASSED! Reached end of test as expected! ---
51 END
```

Listing A.3: Translation of the Tcl-script (listing A.2) into an SpW-script. The initialization of the HPDP, recurring header, and result checks are skipped (indicated with double brackets; the full size is over 500 lines of code)

A.3 Test cases

Test name	Description	Progress	Expected Result	Ref.
GPIO test	Access the board and HPDP chip via SpaceWire and switches on the LEDs which are connected to GPIO pins. The result is observable.	Power Supply, SpaceWire 1 RMAP, GPIO (output)	Written and read values of GPIO direction and GPIO output are equal. *Observable:* All three LEDs are on.	4.4.1
Memories test	Verify the operational capability of the two board memories by writing values to specific addresses. Reading of the same addresses shall return the written values.	Board memories 0 and 1	All read values are equal to the written values. The two external and the internal RAM are addressable.	4.4.1
SpaceWire test	Check the workability of each SpaceWire link (1/2/3). That includes the RMAP module, as well as DMA module.	SpaceWire 1/2/3 RMAP & DMA	All three physical SpaceWire links are available.	4.4.1
Stress test	Write an enormous amount of data to memory 0, 1 and 2 via the SpaceWire links to evince their functionality under stress.	FNC-program can be copied faultless to code memory	An enormous amount of data can correctly be written to the memory via SpaceWire.	4.4.1
Boot test	Load a small FNC-program into memory and start the boot unit FNC 0. The program should activates the LEDs.	The HPDP can boot	FNC 0 boots after giving the corresponding signal and halts after switching on the LEDs.	4.4.1
Response test	Verify that an FNC-PAE can handle a SpaceWire DMA transfer.	FNC $\leftrightarrow$ SpaceWire DMA	FNC 0 can start an SpaceWire DMA transfer that transfers data correctly	4.4.2
Board ID	Read the board ID bitwise from GPIO pin 5. The GPIO pins [4:2] are used to select the corresponding bit of the ID.	GPIO (input), Board ID	The tester receives the right Board-ID	4.4.2
Watchdog	Check the Watchdog interrupt with a very high timer value.	Watchdog	Watchdog status and an interrupt is only set if the watchdog value is reached.	5.3.4

Table A.6: Overview of all test cases (part 1)

Test name	Description	Progress	Expected Result	Ref.
FNC-XBAR	Transfer data and events between both FNC-PAEs via the X-BAR.	FNC 1, FNC-XBAR	Received data and events are equal to the send one.	4.4.2
FNC-TCM	Stress the entire TCM of each FNC.	FNC-TCM	The data read from TCM is equal to the written one.	4.4.2
FNC-Memories	Stress the memories connected to the three memory ports by writing and reading back various pattern.	Board memories 0 / 1 and internal RAM at port 2	The FNC program does not discover a fault, and no single or double bit error is detected by EDAC.	4.4.2
XPP Array func.	Test the basic behaviour of the XPP Array by checking the timing.	XPP Array (partial), Config DMA	XPP Array works functional correct under operational speed.	4.4.2
CDC-FNC-ARY	Stress the clock-domain crossing between the both FNCs and the XPP Array by utilising all ports for a data transfer.	Clock-domain crossing: FNC $\Leftrightarrow$ XPP Array	Received and sent values and events are equal.	4.4.2
RAM-PAE	Verify the functional connectivity of all RAM-PAE objects.	RAM-PAEs	Each memory cell of the RAM-PAEs is functionally connected correctly.	4.3.4
X-RAM	Prove the accessibility of the X-RAM module.	Right and left 4D-DMAs, X-RAM.	The X-RAM is connected right.	4.3.4
X-FIFO	Utilise the Linear DMAs to test the X-FIFO.	Linear DMAs, X-FIFO	X-FIFO works as an expected FIFO queue and stalls if the storage is full.	4.3.4
Stream-IO	Exchange memory content via the Stream-IO ports.	Stream-IO	The Stream-IO ports are available and transferred data is valid.	4.3.4
Full Load	Use as much as possible modules and interfaces at the same time.	Power Supply Voltage	No error occurs while the whole chip is working.	4.4.2

Table A.7: Overview of all test cases (part 2)

List of abbreviations

ALU-PAE Arithmetic Logic Unit Processing Array Element

ASIC Application-Specific Integrated Circuit

BIST Built-in-self-test

BLTS Board Level Test System

BREG Backward Register-object

CDC Clock-domain crossing

DFT Design For Testablity

DMA Direct Memory Access

DSP Digital Signal Processor

EDAC Error Detection and Correction

EEPROM Electrically Erasable Programmable Read-Only Memory

ELF-GCC Executable and Linkable Format GNU Compiler Collection

ESA European Space Agency

FPGA Field Programmable Gate Array

FREG Forward Register-object

FNC-PAE Functional Processing Array Element

GPIO General-purpose input/output

HPDP High Performance Data Processor

HPPM High Performance Processing Module

IP Intellectual Property

JTAG Joint Test Action Group

PLL Phase-locked loop

RAM-PAE Random Access Memory Processor Array Element

RMAP Remote Memory Access Protocol

RTL Register-Transfer Level

SBST Software-based self-testing

SDRAM Synchronous Dynamic Random Access Memory

SFU Specific Functional Unit

SI Signal integrity

SIMD Singel instruction, multiple data

SoC System-on-Chip

SpW SpaceWire

SRAM Static Random-Access Memory

STA Static Timing Analysis

SYSMEM-Arbiter System Memory Arbiter

TCM Tighlty Coupled Memory

Tcl Tool command language

UMDS Universal Data Management System

VHDL VHSIC Hardware Description Language

XPP eXtream Processing Platform

List of Figures

List of Tables

Listings

Bibliography

[AOM06] Alexandre M Amory, Leandro A Oliveira, and Fernando G Moraes. Software-based test for nonprogrammable cores in bus-based system-on-chip architectures. In *VLSI-SOC: From Systems to Chips*, pages 165–179. Springer, 2006.

[Bar14] Stefan Bartels. Investigation of portability of an image processing algorithm on a reconfigurable space-borne parallel processor, 2014. Masterthesis, TU München.

[Bau15] V. Baumgarte. HPDP Demonstrator Chip Backend Specification. Internal Document, Airbus DS, July 2015.

[BBPS06] Alfredo Benso, Alberto Bosio, Paolo Prinetto, and Alessandro Savino. An online software-based self-test framework for microprocessor cores. In *Design and Test of Integrated Systems in Nanoscale Technology, 2006. DTIS 2006. International Conference on*, pages 394–399. IEEE, 2006.

[BCD03] Xiaoliang Bai, Li Chen, and Sujit Dey. Software-based self-test methodology for crosstalk faults in processors. In *High-Level Design Validation and Test Workshop, 2003. Eighth IEEE International*, pages 11–16. IEEE, 2003.

[BEM+03] Volker Baumgarte, Gerd Ehlers, Frank May, Armin Nückel, Martin Vorbach, and Markus Weihnhardt. PACT XPP - A Self-Reconfigurable Data Processing Architecture. *The Journal of Supercomputing*, 26(2):167–184, 2003.

[BHF96] Vamsi Boppana, Ismed Hartanto, and Kent W Fuchs. Full fault dictionary storage based on labeled tree encoding. In *VLSI Test Symposium, 1996., Proceedings of 14th*, pages 174–179. IEEE, 1996.

[BP99] Ken Batcher and Christos Papachristou. Instruction randomization self test for processor cores. In *VLSI Test Symposium, 1999. Proceedings. 17th IEEE*, pages 34–40. IEEE, 1999.

[Bro15] P. Brotzer. HPDP BLTS Proposal. Internal Document, Airbus DS, November 2015. Revision 2.

[CD01] Li Chen and Sujit Dey. Software-based self-testing methodology for processor cores. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(3):369–380, 2001.

[CD02] Li Chen and Sujit Dey. Software-based diagnosis for processors. In *Proceedings of the 39th annual Design Automation Conference*, pages 259–262. ACM, 2002.

[CDS+00] Li Chen, Sujit Dey, Pablo Sanchez, Krishna Sekar, and Ying Cheng. Embedded hardware and software self-testing methodologies for processor cores. In *Proceedings of the 37th Annual Design Automation Conference*, pages 625–630. ACM, 2000.

[CRRD03] Li Chen, Srivaths Ravi, Anand Raghunathan, and Sujit Dey. A scalable software-based self-test methodology for programmable processors. In *Proceedings of the 40th annual Design Automation Conference*, pages 548–553. ACM, 2003.

[ESL01] Mohammed El Shobaki and Lennart Lindh. A hardware and software monitor for high-level system-on-chip verification. In *Quality Electronic Design, 2001 International Symposium on*, pages 56–61. IEEE, 2001.

[GRV03] Olga Goloubeva, M Sonza Reorda, and Massimo Violante. High-level test generation for hardware testing and software validation. In *High-Level Design Validation and Test Workshop, 2003. Eighth IEEE International*, pages 143–148. IEEE, 2003.

[JH11] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.

[KCGP12] Naghmeh Karimi, Krishnendu Chakrabarty, Pallav Gupta, and Srinivas Patil. Test generation for clock-domain crossing faults in integrated circuits. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 406–411. IEEE, 2012.

[KLC+02] Angela Krstic, Wei-Cheng Lai, Kwang-Ting Cheng, Li Chen, and Sujit Dey. Embedded software-based self-test for programmable core-based designs. *IEEE Design & Test of Computers*, 19(4):18–27, 2002.

[KPGZ02] Nektarios Kranitis, A Paschalis, Dimitris Gizopoulos, and Yervant Zorian. Effective software self-test methodology for processor cores. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 592–597. IEEE, 2002.

[KPGZ03] Nektarios Kranitis, Antonis Paschalis, Dimitris Gizopoulos, and Yervant Zorian. Instruction-based self-testing of processor cores. *Journal of Electronic Testing*, 19(2):103–112, 2003.

[Lem10] E. Lemke. HPDP Chip Architectural Design Description. Internal Document, EADS Astrium, October 2010. Issue 2.2.

[Lie10] U. Liebstueck. HPDP Technical Note Debug & BIST Function. Internal Document, EADS Astrium, June 2010. Issue 1.1.

[LKC00] Wei-Cheng Lai, Angela Krstic, and Kwang-Ting Cheng. On testing the path delay faults of a microprocessor using its instruction set. In *VLSI Test Symposium, 2000. Proceedings. 18th IEEE*, pages 15–20. IEEE, 2000.

[LS10] Erik Lembke and Lars Stopfkuchen. HPDP Test and Verification Report. Internal Document, EADS Astrium, October 2010. Issue 2.1.

[PAC] PACT XPP Technologies AG. *XPP-III Reference Manual FNC-PAE Libraries*. Release 1v5.

[PAC08] PACT XPP Technologies AG. XPP-III Components: GPIO, 2008.

[PAC09a] PACT XPP Technologies AG. *Reference Manual: Function-PAE*, November 2009. Release 1v6.

[PAC09b] PACT XPP Technologies AG. XPP-III Components: RAM-IO, 2009.

[PAC09c] PACT XPP Technologies AG. XPP-III Components: Stream-Fifo, 2009.

[PAC09d] PACT XPP Technologies AG. XPP-III Components: Stream-IO, 2009.

[PAC09e] PACT XPP Technologies AG. XPP-III Components: Stream X-Bar, 2009.

[PAC09f] PACT XPP Technologies AG. XPP-III IP for HPDP, February 2009.

[PAC09g] PACT XPP Technologies AG. *XPP-III Programming Tutorial*, February 2009. Version 1.3.

[PAC09h] PACT XPP Technologies AG. *XPP-III Reference Manual: Dataflow Array*, March 2009. Version 1.4.

[PGK+01] Antonis Paschalis, Dimitris Gizopoulos, Nektarios Kranitis, Mihalis Psarakis, and Yervant Zorian. Deterministic software-based self-testing of embedded processor cores. In *Proceedings of the conference on Design, automation and test in Europe*, pages 92–96. IEEE Press, 2001.

[SAH13] Mohsin Syed, Georg Acher, and Tim Helfers. A high performance reliable dataflow based processor for space applications. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 1:1–1:4, New York, NY, USA, 2013. ACM.

[SBR07] Youssef Serrestou, Vincent Beroulle, and Chantal Robach. Impact of hardware emulation on the verification quality improvement. In *Very Large Scale Integration, 2007. VLSI-SoC 2007. IFIP International Conference on*, pages 218–223. IEEE, 2007.

[SH10] Mohsin Syed and Tim and Helfers. HPDP Chip Requirment Specification. Internal Document, EADS Astrium, July 2010. Issue 2.3.

[SHW+08] Mohsin Syed, Tim Helfers, S Weiss, C Schaefer, and M Hahn. On-Board Data Processor for Optical & Microwave Missions Study - Final Report. Internal Document, EADS Astrium, December 2008. Issue 1.0.

[SISF05] Virendra Singh, Michiko Inoue, Kewal K Saluja, and Hideo Fujiwara. Delay fault testing of processor cores in functional mode. *IEICE transactions on information and systems*, 88(3):610–618, 2005.

[Skl76] Joel R. Sklaroff. Redundancy management technique for space shuttle computers. *IBM Journal of Research and Development*, 20(1):20–28, 1976.

[TNF01] Mohammad H Tehranipour, Zainalabedin Navabi, and Seid Mehdi Fakhraie. An efficient bist method for testing of embedded srams. In *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on*, volume 5, pages 73–76. IEEE, 2001.

[Tru15] Diego Andres Suarez Trujillo. Design and implementation of a feature detection algorithm for space debris detection on the high performance data processor (HPDP), 2015. Masterthesis, TU München.

[ZMD98] Yervant Zorian, Erik Jan Marinissen, and Sujit Dey. Testing embedded-core based system chips. In *Test Conference, 1998. Proceedings., International*, pages 130–143. IEEE, 1998.