Nikolaus Georg Sifferlinger, BSc

SmartOS goes Multi-Core: A RISC-V Adventure

MASTER'S THESIS
to achieve the university degree of
Diplom-Ingenieur

submitted to
Graz University of Technology

Supervisor
Dipl.-Ing. Dr. techn. Tobias Scheipel, BSc
Institute of Technical Informatics
Embedded Architectures & Systems Group

Graz, March 2025

The only skill that will be important in the 21st century is the skill of learning new skills. – Peter Ducker

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material that has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present thesis.

Date	Signature

Acknowledgements

A master's thesis is a journey with ups and downs. Therefore, a writer could not complete it without the support of their social environment. The academic supervisor, family, friends, and coworkers must be included in this social environment. Without those people, the burden of a master's thesis could not be carried. Therefore, I have to thank those who always supported me.

First, I want to thank my supervisor **Tobias Scheipel**, who never lost patience with me. Even if my journey took a little bit longer. My supervisor always supported me with ideas and feedback and showed me the right direction.

My Superior **Stefan Rossegger** at work always pushed me to bring this master thesis to an end and gave me the free time from work to end this master thesis.

A special thank you to my girlfriend **Anja Rafajec**, who gave me the strength to bring this master thesis to an end. My family always had faith in me and supported me in the best way possible. I am grateful to my father, **Nikolaus August Sifferlinger**, and my mother, **Ingeburg Sifferlinger**. All these years, they were confident that I would find my way and end my academic studies. My friends gave me the power to carry on.

I will always be thankful to those who gave me the power to end this journey. This master's thesis would not have been possible without you!
Thank you!

Graz, March 2025
Nikolaus Georg Sifferlinger

Abstract

This master thesis is a hardware/software co-design; a RISC-V multi-core system was designed in hardware. The Operating System (OS) SmartOS was extended for this architecture to provide full multi-core OS support and features for Interprocess Communication (IPC).

This master thesis consists of four parts. The first part, **Related Work**, summarizes the knowledge gathered to develop a proper multi-core hardware/software co-design. The second part, **Implemented Work**, presents and explains the implemented multi-core systems and the multi-core versions of SmartOS. The third part, **Measurements and Simulations**, showcases the implemented hardware/software co-design and evaluates its performance. Finally, the last part, **Conclusion**, summarizes the results of this work.

The related work part is separated into a hardware section, a software section and an analysis of the existing multi-core system section. The hardware section shows the state-of-the-art existing bus systems, bus topologies, bus arbiters, and bus data transfer modes, and the systems' benefits and disadvantages are discussed. The software parts show several IPC methods and different multi-core OS approaches and list the pros and cons of the concepts. Further, IPC hardware support features for the RISC-V architecture are shown and explained. At the end of this Chapter, several existing multi-core systems are shown, and their strengths and weaknesses are discussed.

The implementation part demonstrates how the created hardware/software co-design is realized. This section shows the architecture of the RISC-V multi-core system and the new multi-core version of SmartOS. For the hardware, it shows and describes the most significant modules of the multi-core system. The OS shows how the multi-core version is implemented. Therefore, all implemented IPC features are shown and explained.

In the measurement and simulation part, the created multi-core system is analyzed by the resource consumption on the used Field Programmable Gate Array (FPGA). The access time of the implemented crossbar is measured, and the implemented Wishbone (WB) atomic transfer bus mode is shown. The multi-core system's implemented IPC features are measured on the FPGA and explained.

The conclusion is the end of the master thesis. In this chapter, all results are summarized and explained, and possible improvements are shown.

Kurzfassung

Diese Masterarbeit ist ein Hardware/Software Co-Design. Für den Hardware-Teil wurde ein RISC-V-Multi-Core System entwickelt. Für diese Architektur wurde das SmartOS Betriebssystem zu einem Multi-Core unterstützenden Betriebssystem weiterentwickelt und die Features für IPC implementiert.

Diese Masterarbeit ist in vier Teile aufgegliedert. Der erste Teil ist der Theorieteil, dieser zeigt das gesammelte Wissen, das nötig war um das Hardware/Software Co-Design für diese Arbeit zu entwickeln. Teil Zwei zeigt und erklärt das entwickelte Multi-Core-System, sowie die entwickelte Multi-Core fähige SmartOS-Variante. Der dritte Teil ist das Kapitel Messungen und Simulationen. Das Kapitel zeigt Simulationen und Messungen, welche die Performance, des Hardware/Software Co-Designs evaluieren. Der letzte Teil ist die Schlussfolgerung, welche die Ergebnisse dieser Masterarbeit zusammenfasst.

Der Theorieteil ist in einen Hardware-, Software- und existierende Multi-Core-Systemeteil gegliedert. Der Hardware-Teil beschäftigt sich mit den Vor- und Nachteilen von existierenden Bussystemen, Bustopologien und Busübertragungsarten. Der Softwareteil erläutert verschiedene IPC-Konzepte und verschiedene Multi-Core-Betriebssystemarchitekturen und geht im Detail auf deren Schwächen und Stärken ein. Des Weiteren werden die nötigen Features für eine IPC-Hardwareunterstützung für die RISC-V Architektur erläutert. Am Ende dieses Kapitels werden noch verschiedene existierende Multi-Core-Systeme analysiert und ihre Vor- und Nachteile diskutiert.

Das Implementierungskapitel zeigt, wie das Hardware/Software Co-Design umgesetzt wurde. Es zeigt konkret die Architektur vom RISC-V Multi-Core-System und die neue mehrkernfähigen SmartOS Betriebssystem-Version. Im Detail werden bei der Hardwareumsetzung die wichtigsten Bauteile des Multi-Core-Systems erklärt. Der Aufbau und Funktionsweise des Betriebssystems werden erläutert, sowie die entwickelten IPC-Features.

Im Mess- und Simulationskapitel wird die Ressourcennutzung des Multi-Core-Systems auf dem FPGA aufgelistet. Die Zugriffszeit der implementierten Crossbar wurde gemessen und der implementierte WB Atomic Bus Mode wird gezeigt. Am Ende des Kapitels werden die implementierten IPC-Konzepte am FPGA gemessen und erklärt.

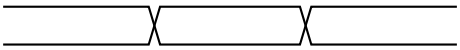
Den Abschluss dieser Arbeit bildet die Zusammenfassung, welche alle Ergebnisse der Arbeit auf einen Punkt bringt und die Ergebnisse diskutiert. Des Weiteren werden mögliche Verbesserungen aufgelistet.

Contents

CHAPTER 1

Introduction

The introduction shows the motivation for this master's thesis, and the research questions are listed. The targets that have to be fulfilled for this thesis are shown. At the end, a general overview of the structure of this thesis is shown.



1.1 Motivation

In times of Internet of Things (IoT), embedded systems are becoming a part of the daily modern life of humans, without even noticing the embedded systems. Thus, the requirements for embedded systems are getting more specific to their use case. Performance enhancement and energy efficiency are two requirements, and a multi-core system improves both. The performance enhancement is achieved by breaking tasks into smaller tasks and running these tasks simultaneously on different cores. The energy efficiency is achieved by running the different cores on a lower frequency than an ordinary one-core system [1]. There, the idea was born to make the next step in the evolution of OS for SmartOS. This step makes SmartOS a multi-core OS.

A best-fitting multi-core architecture must be found and implemented to validate this idea. This architecture should be flexible and robust enough to be reused in the moreMCU [2] project for future research. Therefore, the hardware should be implemented to be quickly adopted. In particular, adding or reducing cores should be done without significant effort to tailor the embedded system to specific use cases.

The key to flexibility in a multi-core system lies in its bus system. A deep dive into the concepts of the bus systems has to be done. Therefore, the analysis of existing bus systems and concepts is needed. To find the optimal bus to fulfil the requirements of a flexible and easy reusable multi-core system.

An embedded multi-core system with a massive amount of cores that can not exchange information between its cores or coordinate the tasks of its core will not be performance efficient. Therefore, the solution is an appropriate IPC concept for the OS. This makes it possible to fully exploit the benefits of more cores.

1.2 Goals

This master's thesis has three main goals. The first target was to evaluate other multi-core systems, bus systems, OS concepts and IPC. This evaluation is divided into the software and hardware parts. The hardware part is to find a proper multi-core concept for SmartOS. The software parts evaluate how SmartOS multi-core-version should be implemented.

Based on the first goal, the other two are built up. The second target is to create a proper hardware multi-core environment. The next listed points should be fulfilled:

- This hardware implementation has to be easily expandable. Therefore, the number of cores should be easily changed.
- System Verilog has to be the used Hardware Description language, as modules already exist in this language.
- The bus arbiter should be real-time capable, as in further usage, real-time is necessary.
- The implemented modules should be easy to use.

The third goal is to implement a multi-core capable version of SmartOS. To make proper usage of the computing power of several cores for the OS possible, IPC features for SmartOS have to be implemented.

To validate the implemented hardware/software co-design, the implemented features should be shown in simulation and tested on the FPGA Artix-7-Nexys-4 [3].

1.3 Thesis Structure and Organisation

The master thesis is structured as follows:

Chapter 2: Related Work shows existing concepts and related work and summarizes all Pros and Cons of the analyzed concepts.

Chapter 3: Implemented Work describes the created multi-core system and implemented multi-core OS in detail.

Chapter 4: Measurement and Simulation evaluates the created hardware/software co-design on a FPGA. This evaluation is made with simulations and measurements via an Oscilloscope.

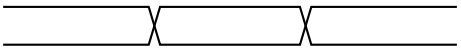
The conclusion of the thesis and possible future improvements are shown in **Chapter 5: Conclusion and Future Work**.

In **Chapter 6: Appendix**, all used publications and definitions for all abbreviations are listed, and the used code for the measurement and simulation part can be found.

CHAPTER 2

Related Work

For this master thesis, knowledge has been gathered to design a flexible multi-core system that supports SmartOS [4], the results are presented in this section. In this section, hardware and software concepts, which are essential for a hardware/software multi-core system design, were analysed and evaluated. Therefore, different bus systems and their supported bus topologies and transfer modes are shown and explained in hardware concepts (Section 2.1). The software part (Section 2.2) shows IPC concepts and different multi-core OS approaches. The pros and cons of all these concepts are discussed. Further, hardware support concepts of the RISC-V for IPC are shown and explained. At the end of the section, related embedded multi-core systems will be shown to learn from their strengths and weaknesses.



2.1 Hardware concepts

The main topic of the hardware concept section is bus systems. To answer the question, "What are the state-of-the-art bus systems?" This section deep-dives into bus topologies, bus arbiters, bus transfer modes, and existing bus systems, discussing the weaknesses and strengths of all these concepts.

2.1.1 Bus topologies

The **way to arrange bus architecture is called bus topology**. The bus topology is significant for how a bus system performs in its use cases. The use case for this master thesis is a multi-core bus system. Therefore, the focus of the evaluation is on this specific case. The benefits and disadvantages of the different bus topologies are shown in this Section. All topologies are explained concerning **complexity**, **performance** and **cost**. The cost term refers to the area a topology needs on a chip or FPGA.

Shared bus topology [5, page 33]

The most straightforward bus architecture is the shared bus. All component cores, peripherals and other components are connected through a single bus. The bus architecture can be observed in Figure 2.1, the bus architecture can be observed. This approach is well suited for small counts of components. However, a considerable number of components will overwhelm the single bus.

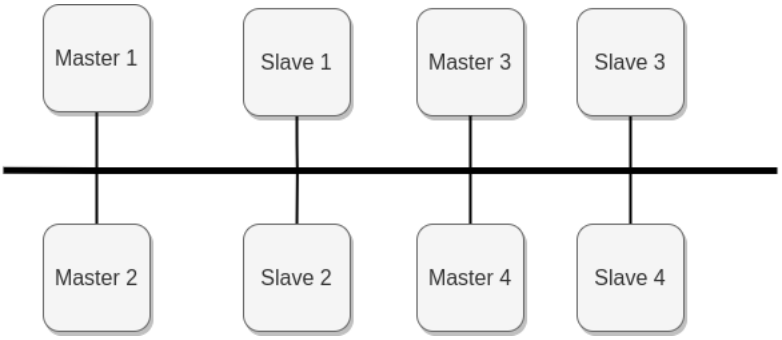


Figure 2.1: Example for a shared bus. Redrawn from [5, page 35].

+ Advantages
    - Simple to implement
    - Small area needed (cost)
- Disadvantages
    - Performance
    - Scalability

Hierarchical bus topology [5, page 33]

The hierarchical topology divides the bus system into several buses connected by a bus interface called a bridge. Therefore, each bus can execute its transfers independently of the others. To clarify, independent refers to the property where all buses can run transfers simultaneously. However, a component from one bus can communicate through the bridge with a component of another bus, thus implying that the buses are not truly independent from each other.

The multiple buses can have different clock frequencies to satisfy the different bandwidth needs of component groups. For example, one bus is responsible for the cores and memory; these components need a higher bandwidth. The other bus could hold the peripherals like Universal Asynchronous Receiver-Transmitter (UART) and General-Purpose Input/Output (GPIO). These components do have a lower bandwidth requirement. Figure 2.2 shows an example schematic for a hierarchical bus with three different buses.

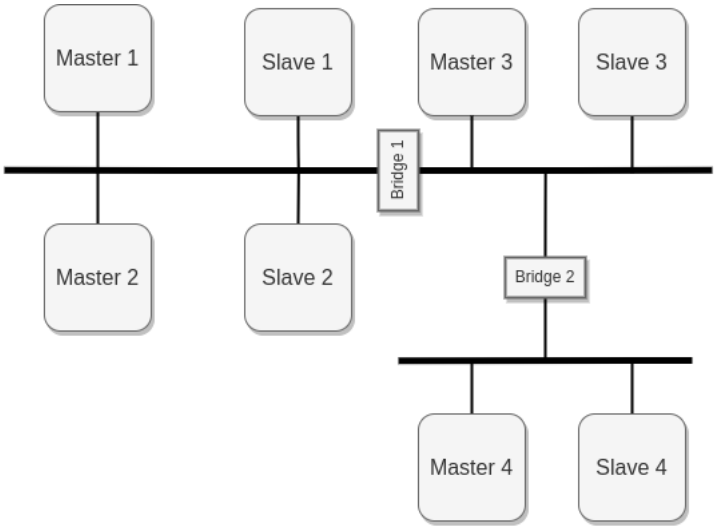


Figure 2.2: Example for a hierarchical bus. Redrawn from [5, page 35].

If communication with components in other buses is needed, the bridges must convert the bus transfers. They must also convert the communication to the different bus frequencies or even bus protocols and buffer it if necessary. These bridges can be quite complex and also add latency to inter-bus communication.

+ Advantages
  - Flexibility to individual bandwidth requirements
  - Average area (cost)
  - Every bus can have its connection at the same time
- Disadvantages
  - Complex bridges

Ring topology [5, page 36]

The ring topology components are connected through one or more buses in a concentric ring form. Figure 2.3 shows an example of a ring bus system with two buses. Bus transfers can be made clockwise or anti-clockwise from one source to a destination. In this topology, an arbiter is needed to ensure correct communication. An Example of an arbiter is the token-passing arbiter. The token passing arbiter defines the component allowed to execute a bus transfer (see Section 2.1.2).

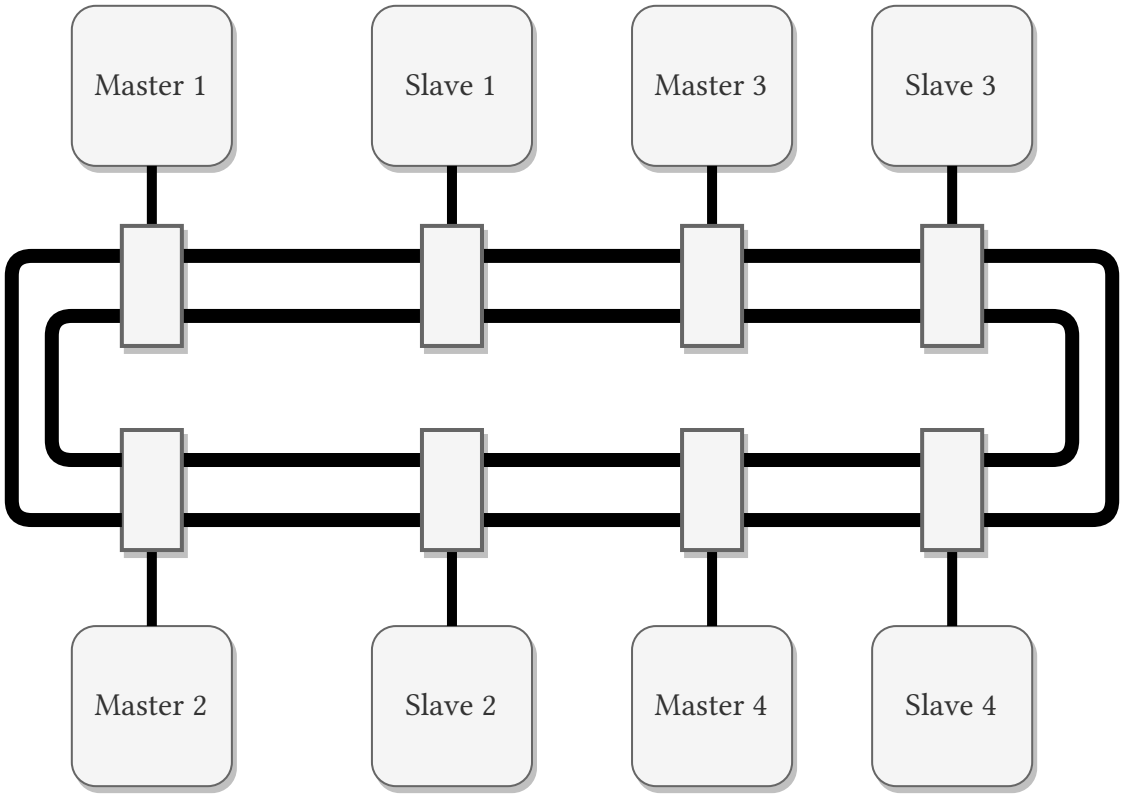


Figure 2.3: Example for a ring bus. Redrawn from [5, page 35].

+ Advantages
  - Medium bandwidth
  - Acceptable bandwidth/area ratio
- Disadvantages
  - Medium area
  - Medium complex implementation

Crossbar switch [6, page 600]

The crossbar switch topology offers the components communication through independent connections. Figure 2.4 shows an example with cores and memory. As can be seen, different ways are possible to create a connection between a core and memory. This connection allows a core to communicate to a desired memory bank, even if there are several other connections, as long as the desired memory is not already occupied.

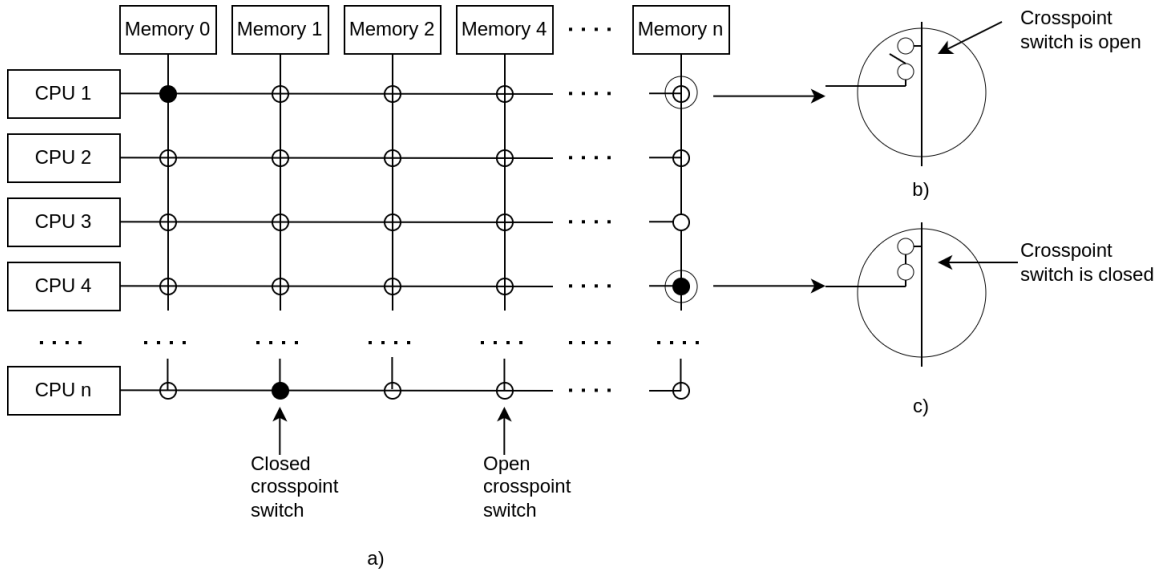


Figure 2.4: Example of a crossbar switch. Redrawn from [6, page 600].

The problem with this approach is that the needed area rises with the number of components to the power of 2. A crossbar switch will be OK for a medium number of cores. However, with many components, the needed area will be overwhelming. Therefore, the trade-off between high costs and bandwidth has always to be considered.

+ Advantages
  - Concurrent bus transfers
  - High bandwidth
- Disadvantages
  - Huge area (cost)
  - Area scales to the power of the square of the number of components

2.1.2 Bus arbiter

The bus system of a multi-core system is a shared resource. An authority managing bus usage is needed to use the bus system effectively. This authority is called the bus arbiter, who decides which master can use the bus. A master requests the arbiter for permission to use the bus, and the arbiter grants or rejects this inquiry based on an algorithm. This algorithm influences the whole system's performance. Therefore, several arbitration techniques are explained and discussed in this Section.

Static fixed priority algorithm [7]

One of the most used algorithms is the static, fixed priority algorithm. In this algorithm, every master (core) has a fixed priority value. If several core requests are made to the arbiter, the core with the highest priority wins. This algorithm does not consider the bandwidth needs of any core. Therefore, a higher-priority core can block the bus, and all lower-priority cores will suffer.

The priority of the running core task can be used instead of a fixed core priority. The running task priorities are used for the arbitration algorithm.

+ Advantages
  - Simple to implement
  - Small chip area
- Disadvantages
  - High-priority cores can block low-priority cores

Time Division Multiplexed algorithm [8]

The Time Division Multiplexed (TDM) algorithm splits the bus usage into time slots. These time slots should have at least the size of a worst-case access time to a component so that at least one bus operation can be done. Every time slot is assigned a fixed characteristic. A possible approach for characteristics is the priorities of the tasks, which run on the different cores. Only a core who runs a task with the correct priority of the time slot can use the bus. The number of time slots equals the number of all possible task priorities. This time slot will repeat periodically.

+ Advantages
  - Simple to implement
  - Small chip area
- Disadvantages
  - If a core does not need a time slot, the bus access time is wasted

Round Robin algorithm [8]

The Round Robin (RR) algorithm is a straightforward approach, where every core has access to the bus in turn for the same quantum of time. All cores share the same latency for the bus access, which could worsen the performance of the running tasks that do not have the exact bandwidth requirements.

+ Advantages
  - Simple to implement
  - Small chip area
- Disadvantages
  - Decreases performance if cores have different bandwidth needs

Lottery algorithm [9]

The Lottery algorithm is based on a randomised arbitration algorithm. This algorithm is called a lottery manager. Every core that wants to use the bus requests the lottery manager. The lottery manager gives the requesting core a ticket. Then, a random ticket is chosen. The choosing process is typically done with a random algorithm. The core with the winning ticket gets access to the bus. The winning ticket grants access to the bus for a defined number of bus cycles. However, to prevent a core from monopolising the bus, the bus cycles are limited. After the granted bus cycles are over, the lottery starts again. To grant cores, which need higher bandwidth and more frequent access to the bus, the number of tickets a core can get can be increased.

+ Advantages
  - Prevents cores from blocking the bus
  - Small chip area
- Disadvantages
  - No deterministic bus usage

Token passing [10]

In ring buses, token passing is used. Therefore, a unique token is used. A component with this token is allowed to use the bus. After the bus operation, the token is passed to the next component in the ring bus.

+ Advantages
  - Simple to implement
  - Small chip area
- Disadvantages
  - Needs ring bus

2.1.3 Bus data transfer modes

Bus architectures have different ways of transferring data via the bus, called bus data transfer modes. A standard bus architecture supports several transfer modes, all optimised for specific purposes. For example, the burst mode transfers a massive amount of data, and the single non-pipelined mode transfers a single data set. All standard bus communication architectures support the basic transfer modes but not all special-purpose modes. These special-purpose modes are proprietary to a single standard bus architecture, but similar special-purpose modes exist among different bus architectures. These modes aim to ensure atomic bus transfer or better bus utilisation.

Single Non-pipelined Transfer [5, page 28-29]

The most straightforward bus transmission is the single data transfer. For this transmission, a master first requests the bus arbiter. If the arbiter grants this request, the master will send the address in the next clock cycle. In the following cycle, depending on the transfer type (read or write), the master will send its write data, or the slave will send the requested read data.

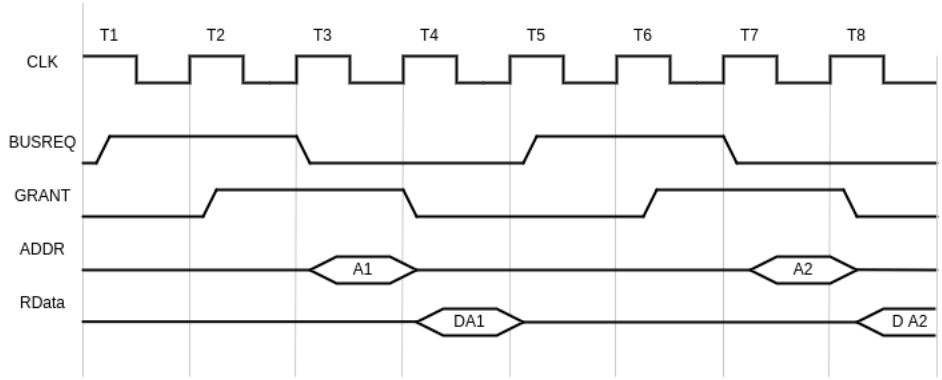


Figure 2.5: Single non-pipelined data transmission mode. Redrawn from [5, page 28].

Figure 2.5 shows an example of two single-read data transfers. The master requests bus usage using the BUSREQ control signal. Then, the arbiter grants this access by sending the GRANT control signal in the subsequent clock cycle. The master notes the GRANT signal and sends the address (A1) of the preferred data in the next cycle. In the subsequent clock cycle, the slave sends the read data (DA1). The slave's response can follow a few cycles later, but for a more precise overview in all examples, the slave sends the answer in the next clock cycle. In Figure 2.5, the second data request follows instantly after the first one. As seen in the figure, reading two addresses took about eight cycles. The next transmission mode will decrease the needed cycle number.

Pipelined Transfer[5, page 28-29]

The pipelined transfer mode reduces the number of clock cycles needed by overlapping the address and data phases of multiple data transfers, significantly improving the bus throughput.

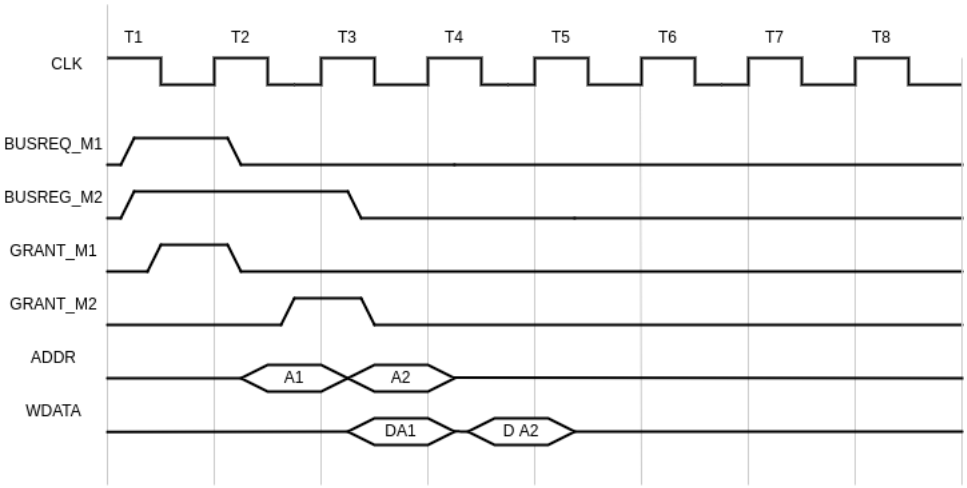


Figure 2.6: Pipelined data transmission mode. Redrawn from [5, page 30].

An example of the pipelined transfer of two masters in the write transmission type can be observed in Figure 2.6. Two masters (M1 and M2) start a separate write data transfer with the corresponding BUSREQ signal in the first clock cycle. The arbiter chooses M1 to have bus access and shows it by sending the GRANT_M1 signal. In the next cycle, M1 sends the address (A1) for the write data, and the arbiter simultaneously sends the GRANT_M2 signal to M2. Therefore, M2 sends the address (A2) of the write data in the subsequent cycle. M1 continues its write transfer within the same cycle by sending the write data (DA1). Finally, M2 ends its write transfer by sending the write data in the next cycle. As seen in the example, the needed clock cycles for data transfers are half compared to the single non-pipelined transfer (see Section 2.1.3).

Burst mode or Block transfer [5, page 29-30]

The burst mode or block transfer solves the problem of a single master making several arbitration requests for multiple data transfers. Therefore, the burst mode allows multiple data transfers within a single bus request.

The burst mode exits in two versions. The simple approach to the burst mode is the non-pipelined burst transfer, as shown in Figure 2.7. This burst mode significantly decreases clock cycles compared to the single non-pipelined transfer (see Section 2.1.3). An improved but more complex version is the burst mode in a pipelined implementation; an example can be seen in Figure 2.8.

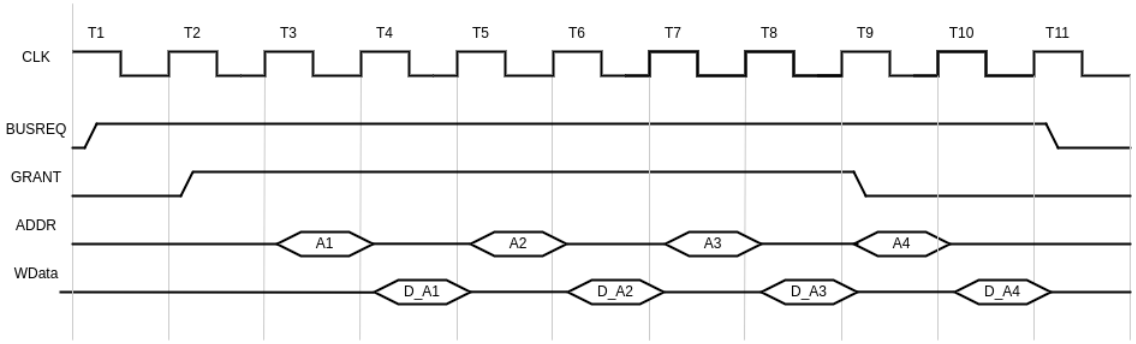


Figure 2.7: Data transmission in burst mode no pipeline. Redrawn from [5, page 31].

Figure 2.7 shows a non-pipelined data transfer. The burst mode concept is explained in a read data transfer example. As seen, the master sends one long BUSREQ signal, and the arbiter approves the access by a GRANT signal, which lasts until the bus sends the final read data address (A4). In the subsequent cycle of the arbiter GRANT signal, the master starts the read transfer by sending the first address (A1). The read data (D_A1) is sent in the next clock cycle, and the first read data transfer is done. Like in the first read data transfer, all other transfers are executed.

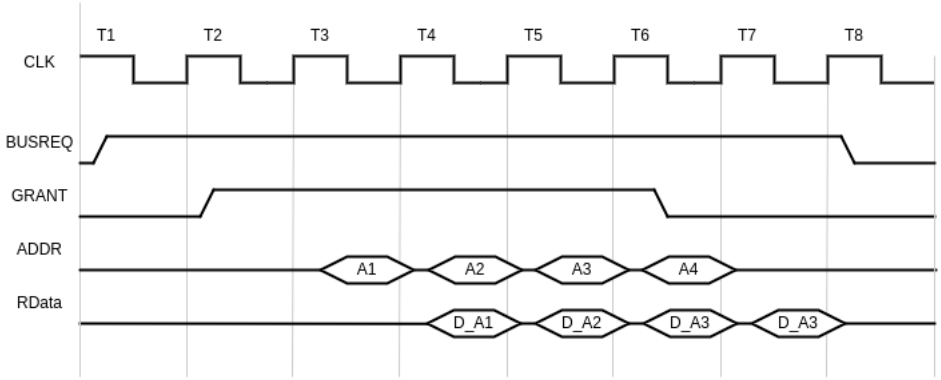


Figure 2.8: Data transmission in burst mode pipelined. Redrawn from [5, page 31].

Figure 2.8 shows the pipelined type of the burst mode. The principle of this mode type is the same as for non-pipelined types, except that the address and read data signals overlap. Thus, overlaps reduce the overall duration by a few clock cycles.

Split Transfer [5, page 32]

Slaves can need several clock cycles to process the read or write data. Therefore, the bus is idle for several cycles, and expensive bus access time is wasted. The split transfer mode solves this problem. This mode allows the idle cycles of the bus to be used by a data transfer from another master.

The split transfer mode works as follows: In a data transfer, a slave can initiate a split transfer by sending the SPLIT response to the arbiter. This happens if the slave's response takes multiple bus clock cycles. As the arbiter receives a split signal from the slave, it prevents the corresponding master from getting further access to the bus. This procedure is called masking the request from a master. The arbiter will now use its arbiter algorithm to choose a new bus user. The masked master will wait until the arbiter signals to continue the masked transfer. This will happen if the slave signals to "un-split" the master. The slave will send this signal when it is ready to transfer data. Then, the arbiter will unmask the master and complete the data transfer.

The split transfer mode increases the bus utilisation, but the arbiter has to be implemented to support slave signals and the split mode.

Read-Modify-Write (RMW) or Atomic Transfer [11]

The RMW is a special transmission mode by the Wishbone bus. This transmission mode is used to explain atomic bus transfer modes, as the Wishbone bus was used for the multi-core system. Similar modes exist for other bus architectures; for Advanced Microcontroller Bus Architecture (AMBA), this mode is called locked transfer, so most bus systems support an atomic mode. The RMW mode is used to explain the concept of this particular type of transmission mode.

The RMW transfer mode guarantees atomic bus transmissions for multiple bus masters. The bus transfer mode makes the bus cycle indivisible. This indivisible term means that master bus transfers using the RMW transfer mode can not be interrupted by another bus master. The bus system is locked, which ensures that only the master can communicate with the slave during this indivisible bus cycle. This transfer mode is needed for the TSL instruction (see Section 2.2.3); it makes these instructions atomic on the bus system.

2.1.4 Bus systems

Avalon bus [5, page 85]

Avalon is a bus system Altera invented, which is now part of Intel. This bus system is a synchronous bus architecture for System-On-a-Programmable-Chip (SOPC) FPGA designs, and it supports the Nios core architecture [12]. The Avalon bus has two standards - the Avalon-memory mapped (MM) and avalon-streaming (ST) standard.

The **Avalon-MM standard** is a bus interface to connect memory-mapped masters and slaves with the crossbar topology. An essential property of the Avalon-MM interface is its reconfigurability. This means Avalon-MM supports different amounts of signals to satisfy different requirements of transfer complexities. The standard supports the burst transfer mode, and more about it can be read in the Avalon Interface Specification [13].

The **Avalon-ST standard** is a bus interface for unidirectional flow of data, which supports Digital Signal Processor (DSP) data, multiplexed streams and packets. The Avalon-ST topology is built as a point-to-point communication bus.

AMBA [5, page 57]

AMBA is a bus standard created by ARM to provide cores with an efficient bus communication standard. An AMBA 4.0 bus system is organised in a hierarchical bus topology. There are commonly several different bus segments in an AMBA system, and a bridge connects these segments. Figure 2.9 shows an example of such a bus system. The AMBA bus system is not exclusive to ARM cores. It supports other core designs as well. There are several versions of the AMBA architectures. The newest version is AMBA 5.0, and this introduces the Coherent Hub Interface (CHI) [14], which supports all bus topologies.

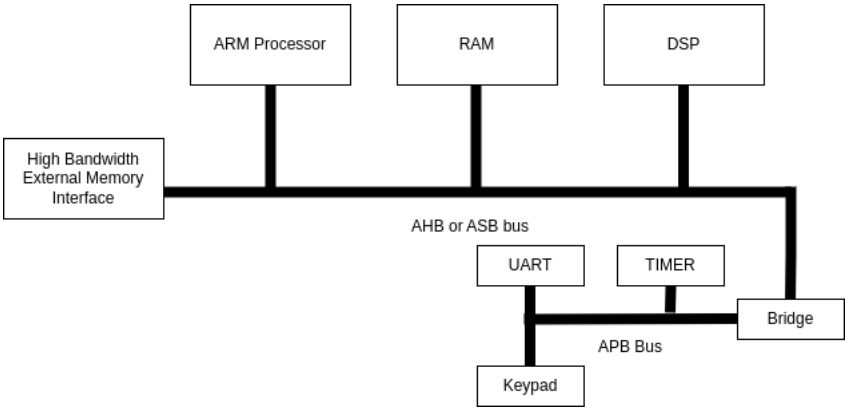


Figure 2.9: An example of an AMBA 4.0 bus system.

The bus AMBA 4.0 is built of segments specialised to different use cases. The most important of these different bus protocols are explained below.

Advanced High-performance Bus (AHB) [15]

The AHB is a high-performance bus that aims to connect high bandwidth needs and high clock frequency components. Therefore, it supports the following bus data transfer modes: burst, split, and pipelined. The AHB is a non-tristate implementation, thus leading to separate read and write data bus lines. The data width could be adjusted to bandwidth requirements in 2^n steps, from 8 to 1024 bits. Commonly, the AHB is used in combination with the Advanced Peripheral Bus (APB) bus, and they are connected through an APB bridge.

APB [16]

The APB is a bus that connects components with low bandwidth requirements. These components can be UART, GPIO or other microcontroller peripherals. It is optimised for low power consumption and reduced interface complexity. Therefore, only the simple single non-pipelined transfer mode is supported. The APB is not used as a single bus system. It is either connected to an Advanced eXtensible Interface (AXI), AHB or Advanced System Bus (ASB) via an APB bridge.

AXI [17]

The AXI standard is for high bandwidth and high performance designs. It is similar to the AHB system but has extended features. One of the features is separate channels for read address, read data, write address, write data and write response. It also provides a transmission mode for atomic operations and supports a matrix bus topology.

Wishbone [11]

The Wishbone bus system is an open-source synchronous bus standard that Opencore had created. The Wishbone bus connects all components and supports multi-master systems. Wishbone splits the components into master and slaves. Typically, cores are masters and peripherals and memory are slaves. The supported transmission modes are single or block read/write transfers and an atomic bus transfer - the RMW cycle. It supports multiple topologies, and the arbitration scheme can be freely chosen.

One of the most valuable features is the ability to define tags for error handling and arbitration algorithms. For instance, the core's running task priority can be sent to the arbitration scheme using user-defined tags.

2.1.5 Comparison and Conclusion of existing bus concepts

Table 2.1 compares the features of a AMBA 4.0, Wishbone and Avalon bus concept. The first viewed feature is **availability**. As an open-source project, Wishbone can be used without any regulations, the other bus system has to be registered. In the topic of **topology**, Whisbone and AXI support all topologies. AMBA 4.0 only supports a hierarchical bus topology, and the Avalon bus supports point to point connections. Regarding **arbitrations**,

all bus systems support different arbitration schemes. The Avalon bus only supports slave-sided arbitration. Wishbone and all AMBA bus systems do support a free choice of the arbitration scheme, except for the APB bus architecture. The AMBA provides a tremendous variation of bus **transfer modes**, but not all sub-architectures support all transfer modes. In the aspect of **bus width** AMBA and Avalon support a greater data bus width but have a smaller address bus range than Wishbone.

		AMBA AHB + APB	AMBA AXI	Avalon	Wishbone
Availability	Open bus system	x	x	x	x
	Registration	x	x	x	
	License				
Topology	Point to Point		x	x	x
	Shared Bus		x		x
	Hierarchical	x	x		
	Ringbus		x		x
	Crossbar switch		x	x	x
Arbiter Algo.	Static Priority	x	x	x	x
	TDM	x	x	x	x
	RR	x	x	x	x
	Token passing	x	x	x	x
Transfer Mode	Single Non-Pipelined	x	x	x	x
	Burst Mode	x	x	x	x
	Pipelining	x	x	x	
	Split Transfer	x	x		
	Atomic Transfer	x	x		x
Buswidth (Bit)	Addressbus	32	32	1-32	1-64
	Databus	8-1024	8-1024	8-1024	8-64

Table 2.1: Comparison of existing bus systems.

Conclusion bus topologies

The bus topologies that fit the design best are highly dependent on the use case. For example, for a low bandwidth-demanding multi-core design with few cores, a shared bus would be fine for the requirements. All concepts trade costs (needed area) and bandwidth. A higher bandwidth can be reached with a greater area, but the scale is not endless. The needed area will reach a point where it will take up too much space in future designs. The large area needed makes the design unfeasible for use cases that need a small design. The crossbar switch, for example, can theoretically deliver a high bandwidth for a massive number of cores, although in practice, the needed area would be far too expensive. Therefore, which bus is used depends on the use case and design requirements.

2.2 Software part

The software part shows and explains the used OS SmartOS and multi-core OS concepts. In Section 2.2.3, the need for synchronisation is explained, several IPC are presented, and their pros and cons are shown. Further, hardware support concepts for the IPC are demonstrated and explained.

2.2.1 OS

SmartOS [4]

The OS that was adopted to a multi-core version is the SmartOS. SmartOS is a Real Time Operating System (RTOS), and the logo of the OS can be seen in Figure 2.10. SmartOS is a microkernel with essential functions like system calls, events, tasks, an internal timeline, resources and interrupt support. The SmartOS is used for research and teaching and mainly supports Microcontroller Unit (MCU) architectures.

Figure 2.10: Logo of SmartOS.

Figure 2.11 shows the strict layering design of SmartOS. The hardware layer handles the peripherals and hardware components of the hardware architecture. This decouples the application software from the hardware architectures. The OS layer contains the SmartOS kernel with all corresponding manager modules. The Application layer includes all user tasks, libraries and all soft-Interrupt Service Routine (ISR).

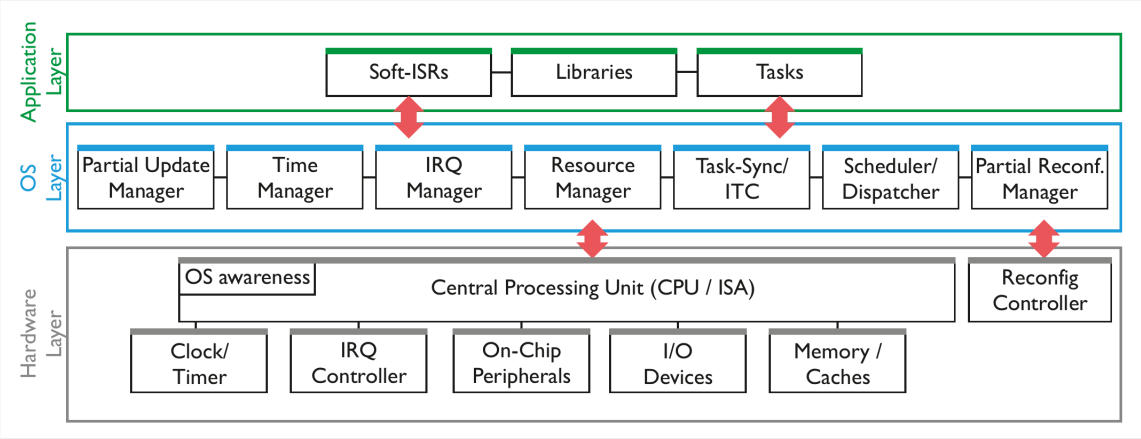


Figure 2.11: SmartOS Layer concept with features [4].

2.2.2 Multi-core operation system concepts

The SmartOS operating system existed in a RISC-V single-core version. This OS was extended to a supporting multi-core OS version. The approaches considered to be used are shown in this Section, along with their benefits and disadvantages. All concepts are drawn in the Figures as one memory for simplicity and viewability. A real multi-core system will have several Random Access Memory (RAM) banks and one or more Read Only Memory (ROM) memory banks. The data of the OS and application will lay in the RAM memory, and the OS code application code and other constants will be saved in the ROM memory.

Each CPU has its own operating system [18, page 531]

The first approach is that every core runs its own OS. A concept schematic can be seen in Figure 2.12. Every core runs the same OS code but has its own OS tables and data structures. Therefore, every system call is handled and executed in the corresponding Central Processing Unit (CPU). Every user task or task is assigned to a fixed core, as every core has task tables. Thus, this could lead to inadequate utilisation of the cores. For example, core one could always be busy because of its tasks, and the other cores can always be in the idle task.

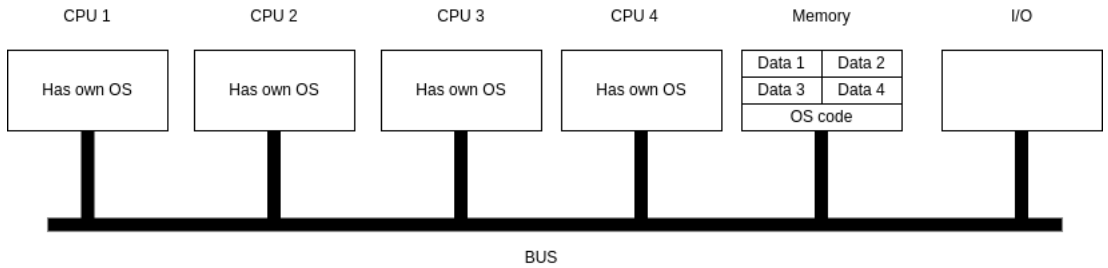


Figure 2.12: Model of a every CPU having an own OS.
Redrawn from [18, page 531].

+ Advantages
  - Simple to implement
  - Comprehensible task for real-time operating systems
  - Scalability
- Disadvantages
  - Tasks are fixed to a core
  - Could lead to bad utilisation of cores
  - Inconsistence OS data between cores

Master-Slave Multiprocessors [18, page 532]

In this concept, one core is the master core, which runs the OS (see Figure 2.13). The master core assigns the tasks to the slave cores. The slave cores ask the master core for a new user task when the currently running task on the core is done. All system calls are redirected to the master core and executed by the master. There is only one version of OS tables and data structures.

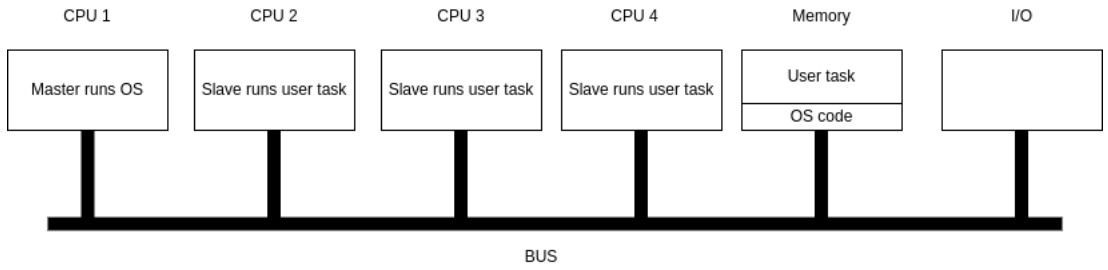


Figure 2.13: Model of a master-slave multi-processor. Redrawn from [18, page 532].

The problem with the concept is that, with a growing number of slaves, the master becomes the bottleneck. The master will spend most of his time handling the slaves' system calls. If the number of slave cores is enormous enough to hold the master in a system call execution, the system will fail.

+ Advantages
  - User task is independent from the core
  - Better utilization among cores
  - No inconsistency between OS data
- Disadvantages
  - Master core is a bottleneck
  - Scalability

Symmetric MultiProcessor [18, page 533]

The Symmetric Multiprocessors (SMP) concept shares one OS among all cores, and the user task can run on every core. To avoid inconsistency of the OS data structure, a synchronisation concept must be implemented if two cores run the OS. The most common approach is that a critical region is secured with a mutex (see Section 2.2.3). Figure 2.14 shows a schematic overview of the concept.

One approach is that the whole OS is one critical region and has to be secured with a lock. This way of implementation is called **big kernel lock**. The problem with this approach is that every system call or any other OS operation of a core blocks all other cores, which try to run the OS. With many cores, the waiting time for executing the OS rises.

The solution for this problem is to split the OS into independent critical regions. A lock mechanism secures every critical region, and different cores can run different parts of the operating system. The problem with this approach is that different critical regions use the same data structures. These data structures are secured with lock mechanisms. If this is not done appropriately, deadlocks can occur.

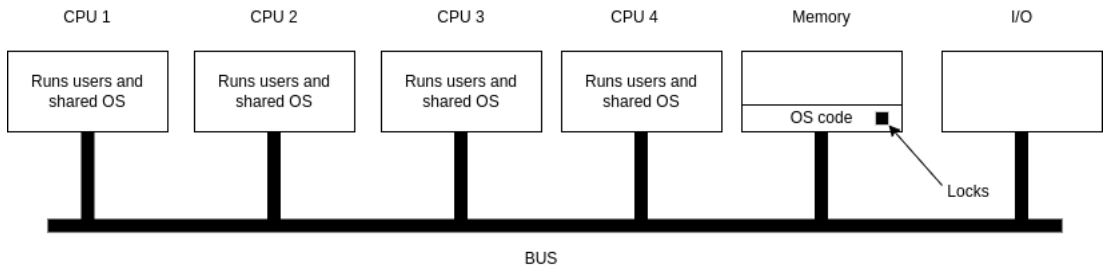


Figure 2.14: Model of a symmetric multi-processor. Redrawn from [18, page 533].

+ Advantages
  - Good Scalability if OS is split into several critical regions
  - No inconsistency between OS data
  - User task is independent from the core
- Disadvantages
  - Possible deadlocks
  - Bad scalability for big kernel lock approach
  - Complex to implement

Conclusion of multi-core operating systems

The shown multi-core OS concepts cover an excellent variation of the possible concepts. The "Every core has its own OS concept" is the simplest of them. However, it shows weaknesses in the consistency of data between the cores. Additional software features for better multi-core data synchronisation must be implemented to eliminate these weaknesses for better multi-core data synchronisation. The master-slave approach has consistent OS data, and the user tasks can run on every core. However, the concept can only support a certain amount of cores, or the master will be overwhelmed by the requests of the slaves.

The SMP approach has good scalability if the OS is divided into several critical regions. However, the synchronisation concepts that must be implemented raise the complexity of this OS type. The best type for a multi-core design highly depends on the multi-core system's preferred use case and size. For this master thesis, every core has its own OS concept was chosen. This OS concept is best fitting for showcasing all implemented IPC concepts.

2.2.3 Synchronisation concepts

General Problem Description [18, page 119]

A central problem for concurrently running tasks is sharing resources, which include time, processors or cores, peripherals, memory, virtual resources, and energy. In a multi-core OS or even in a single-core OS, tasks must communicate with one another or synchronise their execution of shared resources. These synchronisation concepts are called **IPC**, split into three main challenges. The first challenge is passing information among tasks, and the second is preventing two tasks from getting in each other's way. The third challenge is organising the proper sequence of tasks depending on each other.

In an OS, the Scheduler handles the CPU-time of the tasks. Therefore, the Scheduler can interrupt a task depending on the Scheduler's policy. Scheduling away a task that uses a shared resource can lead to errors if the usage of the shared resource is not secured by IPC.

For IPC, three important terms exist, **critical sections**,**race conditions** and **atomic sections** An atomic section is a region in code that can not be interrupted, neither by an ISR nor the OS. That ensures that this atomic section is executed by the CPU in one run and that the memory operations on the bus can not be interrupted by another core.

A **critical regions** or **critical sections** is a part of code where two or multiple tasks use the same resource, data structure or another shared item. For a critical section, one task has to have mutual exclusion to get a correct result. Only one task is allowed to execute the critical region simultaneously. Depending on the OS scheduler algorithm, a task in the critical section can be interrupted, and another Task is scheduled. The mutual exclusion dependence is broken if this task uses the same critical region without a IPC. The action that a scheduler interrupts a task is called preemption. On a multi-core OS, two tasks with the same critical region can run simultaneously on different cores. Therefore, a cross-core synchronisation is needed.

The described problem above is called **race conditions**. Suppose multiple tasks share the same data structures or resources (critical region). Suppose two or more tasks are writing or reading the shared data. The final result depends precisely on who runs when. All interference between two or more tasks that ends in a wrong result is called a race condition. Without a IPC, the probability of a race condition increases with an increased task number, and more cores on those tasks can be executed. As the probability rises that two or multiple use the same critical regions without mutual exclusion. However, four conditions should be fulfilled for an excellent solution to avoiding race conditions.

These conditions are:

1. No task outside the critical regions should block any other task.
2. No assumptions about the speed or number of cores should be made.
3. Only one task should run in the critical region at the same time.
4. No task should wait forever to enter a critical region.

Atomic Regions Implementations

All IPC need the support of atomic regions so that the synchronisation concept is not blocked or interrupted. The implementations for atomic regions differ for single-core and multi-core systems. An easy example is the atomic usage of memory. In a single-core system, preventing the core from executing another code(disabling interrupts) is sufficient to make the memory usage atomic. As in a multi-core system, one or more cores can simultaneously work on the same memory regions; the memory bus must also be atomic to ensure an atomic operation. All implementations of atomic regions need hardware support of the used system.

Disabling Interrupts [18, page 122]

Disabling the interrupts for a code part makes this part an atomic section on the used core, as it can not be interrupted by anything within the core. This is also a primitive to secure a critical region in a single-core OS. The active task cannot be changed or interrupted, so the critical region is safe. Without an interrupt, the preemption of the task in the critical section is secured, and the Scheduler cannot schedule another task on the same core, with the same critical section.

This concept only works if the tasks with the same critical regions are only allowed to run on the same core. For multi-core systems, the disabling interrupts instruction only affects the interrupts on the used core. If another core runs a task with the same critical region, the task can make changes in the critical region without any restrictions. However, this implementation works fine for tasks on the same core and is often used to secure instructions within the OS on one core.

+ Advantages
  - Very easy to implement
  - Secures critical regions within one core
  - Creates atomic region on one core
- Disadvantages
  - No real multi-core synchronisation
  - Tasks with the same critical region fixed to one core
  - Not usable in user tasks

Test and Set Lock (TSL) Instruction [18, page 126]

For multi-core systems or multi-bus masters, an approach is needed that ensures mutual exclusion on memory read and write operations. In this chapter, all IPC need an atomic process for mutual exclusive memory access. The solution for that issue is a hardware-supported lock for the memory operations of the cores like the **TSL instruction**.

The TSL instruction reads the value of the memory word lock into register Rx and stores a nonzero value in the memory variable lock. This memory access is guaranteed to be indivisible. No other core has access to the memory until the instruction is done. The executing CPU running the TSL instruction locks the memory, which is done by locking the bus. It can be observed that this instruction only works with hardware support by the core and bus.

Listing 2.1 shows how this instruction has to be used to secure critical regions. This code is in pseudo assembler code and shows an example implementation. Therefore, two functions must be implemented: the enter_region and the leave_region function. If a task wants to enter a critical region, it has to call the function enter_region.
In this function the TSL instruction is executed on the lock variable. The instruction writes the value of the lock variable into the register and sets the lock variable to 1. The task will enter the critical region if the lock's value is zero. Otherwise, the task will loop the function until the value of the lock variable becomes zero.
When the task has finished the critical region, it will call the leave_region function. This function requires No special instruction to set the lock variable's value to zero in this function.

```
1  /* Example for the use of a TSL instruction in pseudo assembler */
2  // function to enter critical region
3  enter_region:
4     TSL REGISTER,LOCK //copy lock to register and set it to one
5     CMP REGISTER,#0    //check if lock was zero
6     JNE enter_region  //if was not zero loop it again
7     RET               //go back to caller and enter critical region
8  // function to leave critical region
9  leave_region:
10    MOVE LOCK, #0     // set lock to zero
11    RET               // return to caller
```

Listing 2.1: TSL instruction pseudo assembler example, adapted from [18] page 127.

It has to be mentioned that there is an alternative to the TSL instruction, the eXCHanGe (XCHG) instruction. This instruction changes the values of two locations atomically. Both securing critical region concepts work similarly. Therefore, only the TSL implementation is explained in detail.

+ Advantages
  - No strict alternation of tasks
  - Avoids race conditions on multi-core and single-core systems
  - Scalability
  - Guaranteed mutual exclusive memory access
- Disadvantages
  - Busy wait
  - Need for hardware support (core + bus)

Atomic extension for RISC-V [19, page 60 - 63]

In Section, 2.2.3, the TSL instruction is introduced for hardware-supported synchronisation concepts. In this master thesis, a RISC-V-based core was used to implement the multi-core system. Therefore, the RISC-V standard RV32A: Atomic instructions extension was implemented for the used core and bus system.
The RV32A offers two kinds of atomic operations for synchronisation:

- load reserved / store conditional
- Atomic Memory Operations (AMO)

In Figure 2.15, all instructions of this standard can be seen.

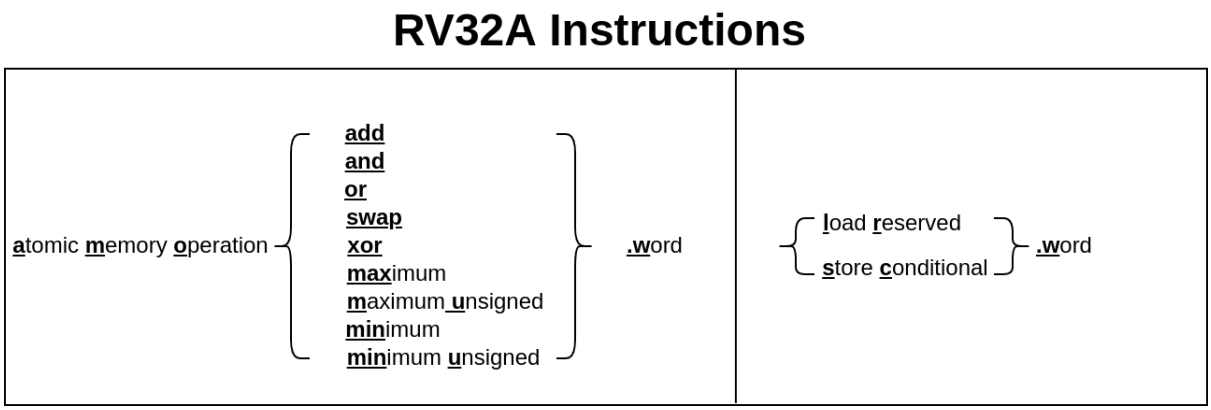


Figure 2.15: Diagram of the RV32A instructions. Redrawn from [19, page 60].

As Figure 2.15 can be seen, there are several atomic instructions. However, the critical instruction is the AMO swap instruction. This instruction is used to implement the test and set concept.

Resource management [20]

A way to guarantee mutual exclusion for resources in an OS is a resource management system. Resource management grants a task temporary and reclusive access to one or more resources. The resource can only be accessed in an interleaved manner. Resource management serialises the tasks when accessing shared resources. The reclusive access to a resource is not preemptive. A task has to allocate a resource before it uses it. This allocation locks the resource from all other tasks. The task has to deallocate the resource if it does not need it any more. Tasks that do not get exclusive access to a resource are suspended and put into a waiting queue. For resource management, these terms are essential:

- Resource **request**
  - A task announces the need for exclusive access, but the access has not been granted yet.
- Resource **allocation**
  - A task gets exclusive access to a resource.
- Resource **deallocation**
  - The exclusive access to a resource is revoked.
- Resource **allocation timeout or deadline**
  - This is the timeout or deadline of a task; before it continues, it works if it does not get exclusive access to the resource.

+ Advantages
  - No Busy wait

- Disadvantages
  - Simple implementation does not avoid deadlocks
  - Complex implementation for multi-core systems

Resource Management Protocols [20]

A problem called **priority inversion** can occur for resource management. Tasks with lower priority that hold a resource can, directly and indirectly, block tasks with higher priority. If the usage of a resource is non-preemptive, a low-priority task blocks a high-priority task that requests the resource. This direct blocking is called **bounded priority inversion**. If, in this situation, a medium-priority task suspends the low-priority task. The low-priority task is prevented from releasing the resource, and the high-priority task is blocked longer than necessary. This indirect block is called **unbounded priority inversion**. Several resource management protocols are presented to prevent priority inversion.

Priority Inheritance Protocol (PIP) [20]

PIP modifies dynamically the priorities of tasks that cause a blocking condition. If a task requests a resource, it is immediately granted if another task does not allocate it. Otherwise, the task is suspended. The task inherits the highest priority of all currently blocked tasks due to resource allocation. Therefore, only if a task with a higher priority requests the resource does the task that owns the resource get the higher priority of the blocked task. If the task deallocates the resource, it returns its prior base priority.

The dynamic priority inheritance leads to three new blocking types.

- **Direct blocking (DB)**
  - A high-priority task that requests a resource allocated by a low-priority task is directly blocked.
- **Push-trough-blocking (PTB)**
  - A medium-priority task is blocked or preempted by a low-priority task, while a resource request of a high-priority task raises the priority of the low-priority task.
- **Chained Blocking (ChB)**
  - If a task is directly blocked by a task that is yet blocked by another task, it is called ChB.

The problem of PIP is that deadlocks can occur if tasks use nested resource allocation.

+ Advantages
  - Avoids unbounded priority inversion
  - Just in time priority change
- Disadvantages
  - No deadlock prevention
  - Possible high blocking delay of high-priority tasks

Priority Ceiling Protocol (PCP) [20]

PCP gives every resource a ceiling priority at the system startup. The ceiling priority is the maximum priority of the resource's tasks. A task can only allocate a resource if another task has not already been allocated and the priority is higher than the ceiling priorities of all of the task's resources already locked by another task. If this is not the case, the task will be suspended. If a low-priority task blocks a high-priority task, it inherits the higher priority. As soon as a resource is released, the priority is readjusted.

+ Advantages
    - Avoids deadlocks
    - Avoids ChB
- Disadvantages
    - Hard to implement
    - Significant computational overhead at runtime

Highest Locker Protocol (HLP) [20]

HLP gives every resource a ceiling priority. The ceiling priority is the maximum priority of the resource's tasks. A task can only allocate a resource if no other task has already allocated it. If a task allocates a resource, it immediately gets the ceiling priority if it is higher than its priority. A task that a resource may not suspend itself. The priority of the task is readjusted if it deallocates the resource.

+ Advantages
    - Avoids deadlocks
    - Avoids ChB
    - Very simple implementation
    - Small runtime overhead
- Disadvantages
    - More unnecessary blocking of tasks

Lock Variables [18, page 123]

The concept of lock variables is a pure software solution, and the idea is to have a shared variable. This variable is zero if no user task claims the corresponding critical region. If a user task tries to enter the critical region, it checks if the lock variable is zero. Therefore, if the lock variable is zero, the task sets the variable to one and enters the critical region. If the variable is one, the task waits until the variable becomes zero again.

The problem with this approach is that it does not avoid race conditions. For example, task 1 reads the variable, and it is zero, and then task 2 is scheduled before task 1 can set the variable to one. Task 2 can enter the critical region and change the lock variable to one. If task one is rescheduled, it sets the variable to one and enters the critical region. The lock variable concept has to be expanded with a race condition avoiding algorithm (see spinlock at Section 2.2.3 for an example) or hardware support to solve this problem.

+ Advantages
  - Can be used before the OS is initialised

- Disadvantages
  - Shifts race conditions problem
  - No multi-core synchronisation
  - Busy wait

Strict Alternation [18, page 124]

The strict alternation concept is a software solution only. It works with an integer variable, the value of which indicates which task's turn it is. All tasks check the variable's value in a loop and run the critical region if it is their turn. This approach is called **spinlock**, and a C code implementation can be seen in the Listing 2.2 below.

```c
/*Example for a spinlock implemenation for 2 tasks in C code */
/*Process 1 spinlock example */
OS_TASKENTRY(task1)
{
   while(TRUE)
   {
      while(turn != 0); // loop until ready
      critical_region();
      turn=1;
      noncritical_region();
   }
}
/*Process 2 spinlock example*/
OS_TASKENTRY(task2)
{
   while(TRUE)
   {
      while(turn != 1); // loop until ready
      critical_region();
      turn=0;
      noncritical_region();
   }
}
```

Listing 2.2: Spinlock code example in C, adapted from [18] page 124.

The initial value of the turn variable is 0, and task 1 executes the critical region first. Task 2 checks the turn variable in a loop until task 1 has done its critical regions and changes the turn value to 1. The tasks constantly check the turn value in a busy wait until the other task

indicates it is their turn. The spinlock implementation can be expanded for more tasks, but the busy waiting time will increase with the growing number of tasks.

The problem with this approach is that one task can block the others even if the task is not in the critical region. For example, if the non-critical region of task 2 is hugely shorter than the non-critical region of task 1, task 1 finishes the critical regions, turns the turn value to 1 and starts the non-critical region. Task 2 now runs the critical regions and changes the value of turn to 0 after finishing the critical region. Task 1 is still in the non-critical region, and task 2 is in its non-critical region. Task 2 has now finished the non-critical region, but task 1 is still in the non-critical region. Task 2 is blocked by task 1, which runs the non-critical code.

+ Advantages
  - Can be used before the OS is initialised
  - Easy to implement
  - Avoids race conditions

- Disadvantages
  - Could block other tasks outside critical regions
  - Busy wait
  - No guaranteed mutual exclusive memory access in multi-core systems

Spinlock in Automotive Open System Architecture (AUTOSAR) [21]

AUTOSAR is a standard framework OS that is established in the automotive and software industry. Therefore, the implementation of spinlock is observed and shown in this Section. Two versions of spinlock exit in AUTOSAR the preemptive and non-preemptive approach. Spinlocks are used for inter-core synchronisation in AUTOSAR.

In Listing, 2.3, the pseudocode for the non-preemptive version of a AUTOSAR spinlock can be seen. This version uses the getSpinLock function to acquire the spinlock. This version is called non-preemptive, as it can not be interrupted on the running core. It disables the interrupts, and therefore, the scheduler can not change the running task.

```
1 //Non-preemptable spinlock in AUTOSAR
2 DisableAllInterrupts()
3 GetSpinLock(lock)
4  //critical section
5 ReleaseSpinLock(lock)
6 EnableAllInterupts()
```

Listing 2.3: Non-preemptive AUTOSAR spin lock in pseudo code, adapted from [21] page 3.

Listing 2.4 shows the preemptive version of the spinlock. It uses the TryToGetSpinLock function to lock the spinlock. This function returns a success if the spinlock was locked successfully. If the lock is successful, the critical regions are executed; otherwise, it enables the interrupts again and starts again from line 1. This gives the scheduler the chance to execute another task.

```
1 //Preemptable unorded spinlock in AUTOSAR
2 DisableAllInterrupts()
3 if TryToGetSpinLock(lock) != TRYTOGETSPINLOCK_SUCCESS
4 then
5    EnableAllInterupts()
6    got to line 1
7 //critical section
8 ReleaseSpinLock(lock)
9 EnableAllInterupts()
```

Listing 2.4: Preemptive AUTOSAR spin lock in pseudo code, adapted from [21] page 4.

Peterson's Solution [18, page 125]

Peterson's Solution is a software-only approach that achieves mutual exclusion without strict alternation. Peterson's algorithm consists of two functions, which all tasks with the same critical region must use. An example in C code for the two functions can be seen in Listing 2.5.

The code shown below is an implementation for two tasks. Before entering a critical region, every task has to call the enter_region function. In this function, the task indicates its interest to run the critical region by writing a TRUE value into the interested array. The task changes the turn variable to its task number. The while loop is not executed if no other task is interested (see row 16 in Listing 2.5). The interested task now executes the critical region, then calls the leave_region and changes the value in the interested array to FALSE.

If two tasks want to enter the critical region simultaneously, the task that last writes its task number into the turn variable has to wait. For example, task 0 first enters the enter_region function and writes to the interested array and the turn variable. Task 1 does the same but after task 0. Therefore, task 0 immediately returns in the while loop, and task 1 loops until task 0 calls the leave_region function. Then, task 1 can enter the critical regions and call the leave_region function.

```c
/* Example for a Peterson's Solution in C code */
#define FALSE  0
#define TRUE   1
#define N      2   /*number of tasks*/

int turn;            /* whose turn is it */
int interested[N]    /*all values intially 0(FALSE) */

void enter_region(int task) //task is 0 or 1
{
   int other;                 //number of othertasks
   other= 1 - task;           //computer other task
   interested[task] = TRUE;   //show that task is interested
   turn process;              //set flag
   //do nothing until nobody is interested
   while(turn == task && interested[other] == TRUE); //busy wait
}

void leave_region(int task) //task leaves critical region
{
   interested[task] = False; //indicate leaving of critical region
}
```

Listing 2.5: Peterson's solution example, adapted from [18] page 125

+ Advantages
  - No strict alternation of tasks
  - Avoids race conditions

- Disadvantages
  - Busy wait
  - Tasks in the non-critical region can block other tasks
  - Busy waiting time increases with a growing number of task
  - No guaranteed mutual exclusive memory access in multi-core systems

Semaphores [22, page 245]

All previously explained concepts worked with a busy wait. The following concepts will eliminate this disadvantage. The first concept is the semaphores, which coordinate the operations of different tasks. The fundamental principle of semaphores is that two or more tasks cooperate with a method of simple signals.
These signals work so that one task can be stopped until it receives a particular signal from another. A complex coordination requirement could be fulfilled with an appropriate

structure of the signals. The semaphore structure has to be manipulated only with the following atomic operations:

- Initialization of the count
- The SemWait function
- The SemSignal function

In Listing 2.6, an example of the needed signal functions and semaphore structure can be observed. For simplicity, the yield or wake-up task functions are not implemented and are only shown as comments. The struct semaphore holds (see line 2) a count and the queue for the waiting tasks. The semaphore count should be initialised with a non-negative integer value. For this example, the value is zero.

```
1  /* Example for a semaphore implementation in C code*/
2  sturct semaphore
3  {
4     int count;
5     queueType queue;
6  }
7
8  void semWait(semaphore sem)
9  {
10    sem.count--;//has to be atomic
11
12    if(sem.count < 0)
13    {
14       //place the process in sem.queue;
15       //suspend (set to sleep) this process
16    }
17 }
18 void semSignal(semaphore sem)
19 {
20    sem.count++;// should be done with hardware support
21    if(sem.count <= 0)
22    {
23       //remove a process X from sem.queue;
24       //set process x to ready
25
26    }
27 }
```

Listing 2.6: Semaphore example in C code, adapted from [22] page 246.

The function **semWait** operation decrements the semaphore count, and if the count is smaller than zero, no signal is sent. In this case, the running task places itself in the semaphore queue and suspends itself. If the count equals or exceeds zero, a signal was sent before, and the task continues running.

The function **semSignal** increments the semaphore count and checks if other tasks are waiting for the signal. A task is waiting in the semaphore queue if the counter is smaller or equal to zero. The task should be put in the ready list of the OS and removed from the semaphore queue.

```
1  /* Example for a binary semaphore implementation in pseudo C code*/
2  sturct binary_semaphore
3  {
4     enum{zero,one} value;
5     queueType queue;
6  }
7
8  void semWait(semaphore sem)
9  {
10    if(sem.value == one)
11       sem.value == zero;
12    else
13    {
14       //place the process in sem.queue;
15       //suspend (set to sleep) this process
16    }
17 }
18 void semSignal(semaphore sem)
19 {
20    if(sem.queue is empty())
21       sem.value == one;
22    else
23    {
24       //remove a process X from sem.queue;
25       //put process x into ready qeue of the OS
26    }
27 }
```

Listing 2.7: Binary semaphore example in pseudo C code, adapted from [22] page 247.

An adaption of semaphores for securing critical regions exists, the so-called binary semaphore. One difference between a semaphore and a binary semaphore is that the “count” of the binary semaphore is a binary variable. The value is zero or one, therefore it is called binary. The exact task must call both functions to secure a critical section with a semaphore. An example of a binary semaphore can be seen in Listing 2.7.

+ Advantages
  - No strict alternation of tasks
  - Avoids race conditions on multi-core and single-core systems
  - Scalability
  - Guaranteed mutual exclusive memory access
  - No busy-wait

- Disadvantages
  - Need for hardware support
  - Simpler solutions for securing critical regions are possible

Mutual Exclusion (mutex) lock [18, page 132]

The mutex lock is a simpler version of binary semaphore, and its use case is to secure critical regions. The mutex is a shared variable with the two states locked and unlocked. The locked value of the variable is one, and the unlocked value is zero.
To guarantee mutual exclusion on memory access, an atomic instruction (see in section 2.2.3) in the lock procedure is mandatory. Otherwise, other tasks can interrupt this memory access and corrupt it. In a mutex, two procedures are used, which can be seen in Listing 2.8.

```
1  /*Example for a mutex implementation in pseudo assembler*/
2  //function for locking a mutex
3  mutex_lock:
4        TSL Register,MUTEX      // copy  mutex to register and set it
             to 1
5        CMP REGISTER,#0         // check if mutex was zero
6        JZE ok                  // if mutex was zero return and enter
             critical region
7        CALL thread_yield       // yield task if mutex was already
             locked
8        JMP mutex_lock          // retry
9  ok:   RET                     // return and enter critical region
10 //function for unlocking a mutex
11 mutex_unlock:
12       MOVE MUTEX,#0           // unlock mutex with zero
13       RET                     // return and leave critical region
```

Listing 2.8: Mutex example in pseudo assembler, adapted from [18] page 133.

The first function is **mutex_lock**. It has to be executed before entering a critical region. In this function, the task writes the mutex value to the register and sets the value of the mutex to 1. If the value of the mutex was zero, the mutex was unlocked, and no other task ran the

critical section. The running task will return from the function and execute the critical section.
Otherwise, the task will yield itself until the Scheduler will rerun it. When the task is rescheduled, it will run this code in a loop until the mutex is unlocked. This looks like a busy wait, but other tasks can be scheduled with the yield function.

After finishing the critical region, the running task has to call the **mutex_unlock**. In this function, the value of the mutex will be set to zero, and the mutex is unlocked. No special instruction is needed for this operation because it is a single write memory operation.

There are mutex implementations with an additional interface, the **mutex_trylock** function. This function tries to lock the mutex. A failure code is returned if the lock fails instead of yielding the task. Then, the task can decide what it should do next.
The mutex concept could be implemented in the user space of an OS. The checks for the mutex value do not need any kernel support, and only sleep or yield functions are system calls. Therefore, it is easy and efficient to implement. It is essential to mention that nested mutex could easily trigger deadlocks if the implementation is not done carefully.

+ Advantages
    - No strict alternation of tasks
    - Avoids race conditions on multi-core and single-core systems
    - Scalability
    - Guaranteed mutual exclusive memory access
    - No busy-wait

- Disadvantages
    - Need for initialised OS
    - Possible deadlocks

Monitors [22, page 257]

Monitors are a programming language construct that provides functionality similar to a semaphore. However, monitors are much easier to control than semaphores. The observed monitor concept is the signal approach. A monitor is a software construct that consists of an initialisation sequence, local data, and one or more procedures.

The internal procedures are the only ones with access to the monitor's local data. External procedures are not allowed to manipulate or even read the local data. A task can only enter the monitor using an internal procedure; no other entrance is allowed. It is mandatory that only one task at a time can be entered into the monitor. All other tasks must be blocked and wait until the monitor is free.

In the monitor, a synchronisation concept has to be implemented to satisfy the need for useful concurrent processing. The chosen synchronisation concept of this monitor approach is condition variables. These condition variables have to be only accessible from the internal of the monitor. The two interfaces of the conditions variables are:

1. **cwait(c)**: This function suspends the task until conditions c is fulfilled. Other tasks can now be entered into the monitor because it is free again.
2. **csignal(c)**: This function signals that condition c is fulfilled and a blocked task can continue its execution. If no task is waiting, nothing happens.

In Figure 2.16, the structure of the monitor software module can be seen. For a better overview, only one entrance of the monitor is drawn, but in reality, a task can enter the monitor by invoking any of the procedures. The entrance is guarded, and only one task can enter the monitor simultaneously. If a task in the monitor waits for condition x, it calls the **cwait(x)** function and will be added to the condition x waiting queue. The task leaves the monitor and will wait to re-enter the monitor until another task calls **csignal(x)**. The other task will call **csignal(x)** if it recognises that the condition x has changed.

+ Advantages
  - Avoids race conditions
  - Scalability
  - No busy-wait
  - Guaranteed mutual exclusive memory access
- Disadvantages
  - Complex software module
  - Conditions variables have to be implemented

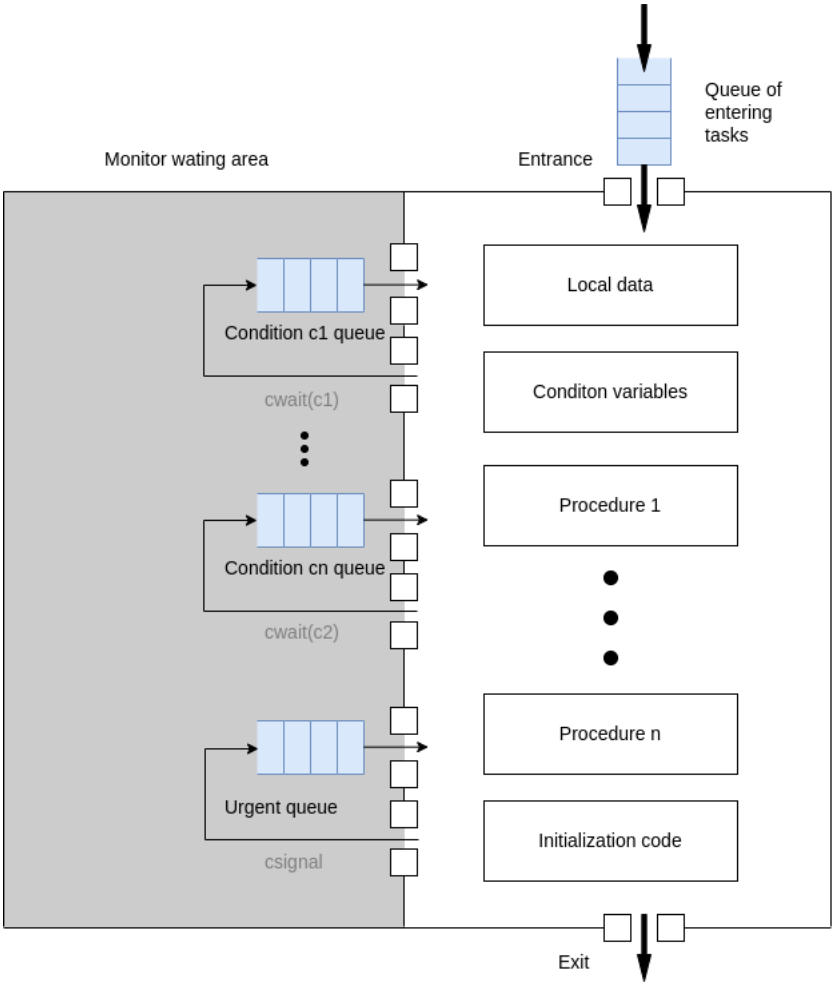


Figure 2.16: Structure of a monitor. Redrawn from [22, page 259].

Message Passing [22, page 263-279]

A good OS provides a solution for sharing information among the tasks or cores. A solution for this problem is the message-passing concept. Message-passing systems exist in many forms. However, typically, they share the same base structure. The base structure commonly has these two functions:

- send (destination, message)
- receive (source, message)

This procedure provides the minimum functionality for message-passing tasks. A task uses the send function to pass data as a message to another defined destination task. A task that wants to get information from another task calls the receiving function. The information is the source and the message.

For a message-passing system, several design issues have to be observed. Table 2.2 shows an overview of these design issues.

- **Synchronisation**
  - Send
    - blocking
    - non-blocking
  - Receive
    - blocking
    - non-blocking
    - test for arrival
- **Message format**
  - Content
  - Length
    - fixed
    - variable
- **Message addressing**
  - Direct
    - send
    - receive
      - explicit
      - implicit
  - Indirect
    - static
    - dynamic
    - ownership
- **Queuing discipline**
  - First In First Out (FIFO)
  - Priority

Table 2.2: Overview of message passing design issues, adapted from [22, page 264].

Synchronisation

The exchange of messages between two tasks implies some synchronisation. Without this synchronisation, data can be corrupted in case of race conditions. Therefore, three common combinations of synchronisation concepts are shown.

1. **Blocking send, blocking receive:**

   The sender and receiver are blocked until the message has arrived. The sender puts the message into the message queue and waits until the message is consumed. The receiver checks the message queue until a message is received and consumes the message. When the message is consumed, both tasks continue. This concept is used to ensure a tight synchronisation between the corresponding tasks.

2. **Non-blocking send, blocking receive:**

   The sender only puts the message in the message queue and continues its work, and only the receiver has to wait for a requested message. The receiver checks the message queue until it receives a message and only continues its other work if

a message is received. This concept is the most practical solution. The receiver task needs the information of the sender's message before its work can deliver a useful solution. The sender task can provide several messages to other tasks, which rely on this message for correct execution.

3. **Non-blocking send, non-blocking receive:**

   Both involved tasks do not have to wait and can continue their execution. The sender puts the message into the message queue and continues. The receiver checks if a message is available and either processes it or continues its work.

Addressing

The design issue on how the send function specifies which task should receive the message is called addressing. Addressing is categorised into two groups: direct and indirect addressing. In the **direct addressing** scheme, the send function includes identifier information of the destination task. The receiving task must know which task to expect a message from. This is an effective way to cooperate with concurrent tasks.

The **indirect addressing** uses a shared data structure to distribute messages. If this shared data structure is a queue, these are generally called mailboxes. The sending process puts the message into the mailbox, and the receiving task picks it up from the mailbox.

The great benefit of this approach is the decoupling of the sender and receiver. That offers greater flexibility for the usage of messages. Therefore, different relationship schemes between sender and receiver are possible. The possible relationship schemes can be seen in Figure 2.16.

The **one to one** relationship is for private communication between two tasks. Their communication is free from erroneous interferences from other tasks. The mailbox is often called a port in the **many to one** relationship. The use case of the relationship scheme is the client/server interaction. The **one to many** relationship scheme allows one task to broadcast a message to multiple receiver tasks. The **many to many** is the last relationship scheme. It is used to provide concurrent multiple servers to multiple clients.

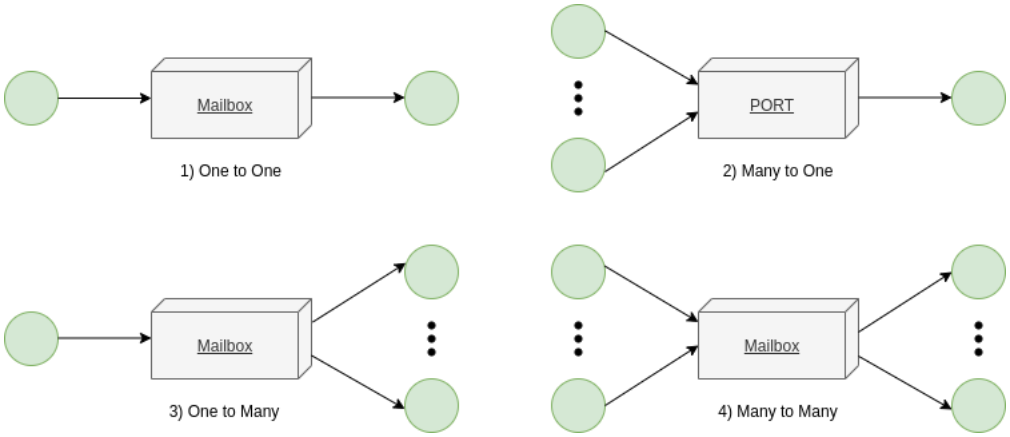


Figure 2.17: Indirect Addressing types. Redrawn from [22, page 267].

Message Format

The message format highly depends on which environment is used and the purpose of the message facility. If a fixed or variable message length is used, it depends on the OS design choice. This choice is influenced by the terms of processing time and storage overhead.

Figure 2.18 shows an example of a variable-length message. However, the basic structure of a message will always have two parts: the header and the body. In the body lays the message information. The header contains the message's meta information. This meta information can be the message type, destination task's ID, sender task's ID, optional message length and control information. The control information can be a sequence number, priority files or other meta information.

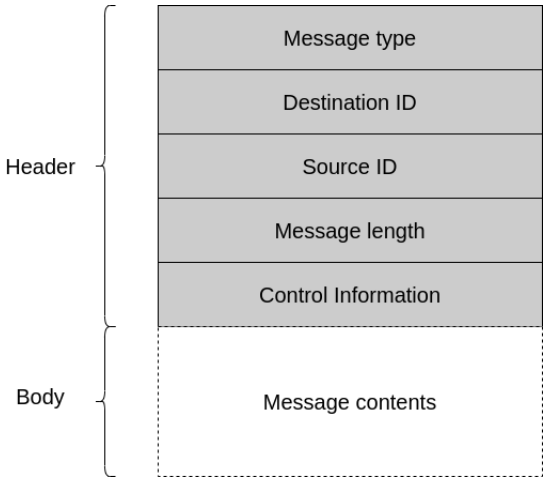


Figure 2.18: General example for a message format. Redrawn from [22, page 267].

Queuing Discipline

Queuing discipline is when a receiver chooses which message is processed first. The easiest solution is the FIFO principle. The first incoming message in the mailbox is the first that is picked by the receiver task. Another principle is that messages have a priority; on this, the first message is chosen. The last principle is to let the receiver choose which message to pick up.

+ Advantages
    - Avoids race conditions
    - Scalability
    - No busy-wait
    - Supports information exchange between tasks
- Disadvantages
    - Complex approach for securing critical regions

Conclusion IPC concepts

Every previously shown IPC concept does cover one or more issues of the IPC (see Section 2.2.3). Resource management protocols, lock variables, strict alternation, Peterson's solution and mutex only cover the avoiding race condition term. All of these concepts need hardware support for atomic operation within the core and atomic bus operations.

Disabling Interrupts makes sections atomic and can be used to secure critical sections on the same core. The TSL instruction is more of a proper hardware support type than a synchronisation concept and has to be used in mutex, semaphores, message passing, and all other concepts to achieve atomic memory read/writes. Without an atomic implementation, all concepts can fail, as they can be interrupted, and the data can be corrupted.

The strict alternation does avoid race conditions without an initialised OS, as tasks can not be suspended before this initialisation of the OS is done. On the other hand, the busy waiting times for tasks are too long for the most common usages. The mutex concept is a spartan hardware-supported race conditions-avoiding concept. It does not have a busy wait. Therefore, it is a good solution for avoiding race conditions when the initialisation of the OS is done. Semaphores can be used to avoid race conditions and to synchronise the proper sequences of tasks. It is a more complex approach for securing critical regions than the mutex, although semaphores cover more IPC issues.

Message passing could be used for all three issues of the IPC, although its main focus is communications between tasks. On a good developed OS, the variation of synchronisation concepts should cover all three issues of IPC.

2.3 Related multi-core systems and concepts

2.3.1 Common classes of multi-core systems

Overview of multi-core classes [23, page 371-373]

There are three classes of multi-core systems: shared-memory multi-core systems, clusters, and warehouse-scale computers. Cluster and warehouse-scale computers look like individual computers connected by a network. The processors in these systems cannot access the memory of other processors without the support of software protocols running on both processors. This work focuses on shared-memory multi-processors or shared-memory multi-core systems and their subclasses. The cluster and warehouse-scale computers will not be discussed and are only mentioned for the big picture.

The shared-memory multi-processor systems are grouped by their memory organisation. The reason is that systems can not only be defined by the number of cores as what is a large or small number of processors tends to change over time. The first group is called SMP or centralized shared-memory multi-processor; the other architecture is called Distributed Shared Memory Multiprocessor (DSMP). The symmetric term in this class refers to the fact that the address space is shared.

SMP class [23, page 371]

The SMP class generally has 32 or fewer cores. That makes it possible to share a sole centralised memory along all processors. All processors have equal access to this memory. Thus, this is where the term symmetric comes from. SMP systems have a uniform worst-case latency from memory along all cores as they have the same distance between the memory and the core. Therefore, SMP are called Uniform Memory access (UMA) multi-processors.

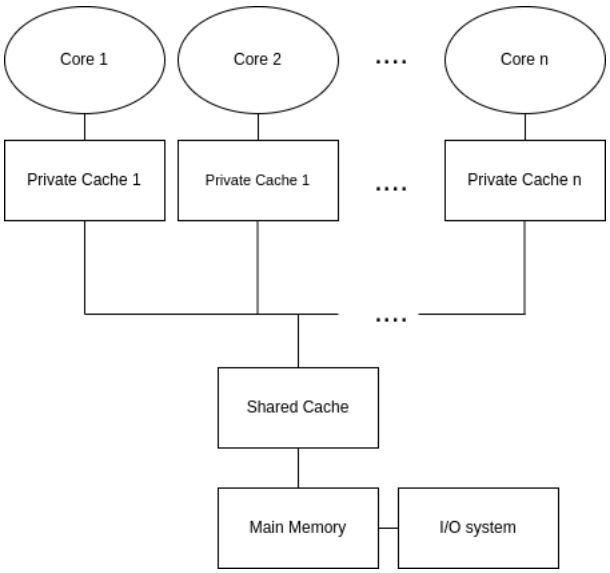


Figure 2.19: Example of a SMP or UMA system.
Redrawn from [23, page 372].

Figure 2.19 shows a typical example of SMP architecture. The cores commonly have one or more levels of private cache. The cores have mostly one shared cache, but always one shared main memory. It is also possible that the cores have no caches and the memory access is handled by Direct Memory Access (DMA) controller. The interconnection network connects the cores and the memory in a multiple-chip design; in a single-chip design, the interconnection network is just the memory bus.

The advantages and disadvantages of the SMP class are listed below.

+ Advantages
  - Uniform latency to all memory
  - Simpler for software and programmer
- Disadvantages
  - Scalability
  - Lower memory bandwidth with a growing number of cores
  - Increasing memory latency with a growing number of cores

DSMP class [23, page 372]

The DSMP architecture is defined by the reference that it has generally between 16 and 64 processor cores. It has to be mentioned that with increasing core performance and the consequential need of the core's memory bandwidth requirements, the number of preferred processor cores for DSMP systems tends to shrink.

Figure 2.20 shows an example for a DSMP system. The multi-cores have a local memory, but the processors share all memories through the interconnection network. The fact that the access time of the data word depends on the location in memory is called Nonuniform Memory Access (NUMA). A high-bandwidth interconnect is recommended to avoid a bottleneck between the multi-core systems.

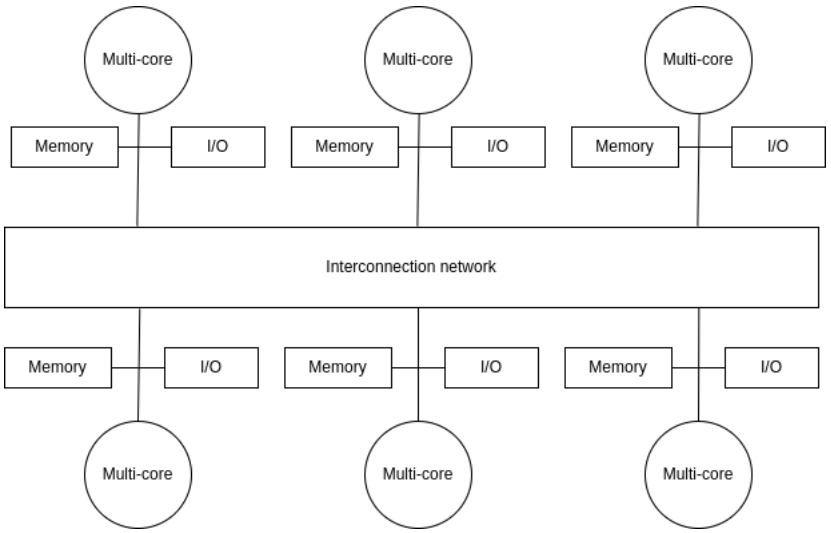


Figure 2.20: Example of a DSMP or NUMA system.
Redrawn from [23, page 373].

The advantages and disadvantages of the DSMP class are listed in the enumeration below.

+ Advantages
  - Increase of bandwidth to local memory
  - Reduce of latency to local memory
  - Scalability

- Disadvantages
  - communication between processors is more complex
  - Complex software to take advantage of increased memory bandwidth
  - latency depends on the location of the data word

2.3.2 Existing Multi-core system with RI5CY cores

For the SmartOS, a RI5CY CV32E40P single-core architecture version exited, so RI5CY CV32E40P was chosen as the architecture. This section observes several existing multi-core systems with similar cores to design an appropriate multi-core system.

Fulmine [24]

The Fulmine design is a multi-core system System on Chip (SoC) architecture, whose usage is near sensor data analytics for IoT endpoints. Its design goal was to provide a low-energy hardware-accelerated solution for deep Convolutional Neural Network (CNN) and encryptions operations. Two frequency-locked loops (FLL) provide different clock frequency domains to achieve the low energy requirement. These domains have different external voltage supplies to power-gate them independently.

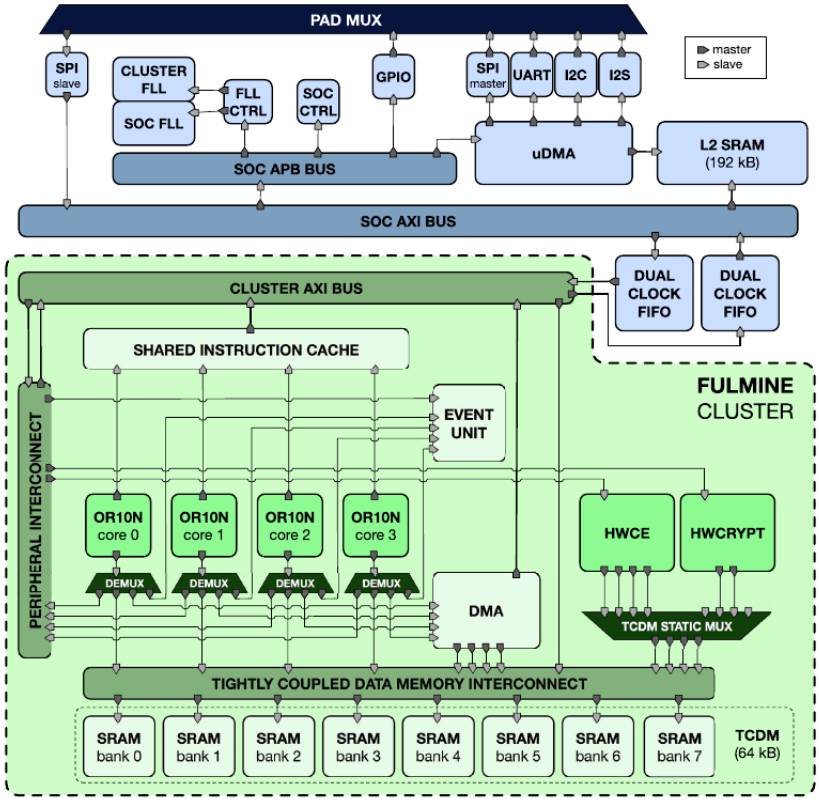


Figure 2.21: Architecture of Fulmine, Source [24].

In Figure 2.21, the architecture of the Fulmine system can be seen. As someone can see, the design contains a Fulmine cluster and peripherals, which are all connected through a SoC AXI bus. The peripherals have their own APB bus, which connects them to the SoC AXI bus. As it can be observed, the used bus system is the hierarchical AMBA approach.

The cluster's internal cluster AXI bus is connected through the dual clock FIFO units to the central SoC AXI bus. The reasons for this are the two different clock frequency domains. The Fulmine cluster is made of several general-purpose cores, Tightly Coupled Data Memory (TCDM) banks, a shared instruction cache, a DMA controller, an event unit, hardware acceleration units for encryption and CNN and several interconnects.

Mia Wallace [25]

The Mia Wallace design is a multi-core system for near-sensor processing edge not in the IoT. This work focuses on the hardware accelerated CNN processing. The software-based CNN algorithm can be executed power-saving through the hardware acceleration units.

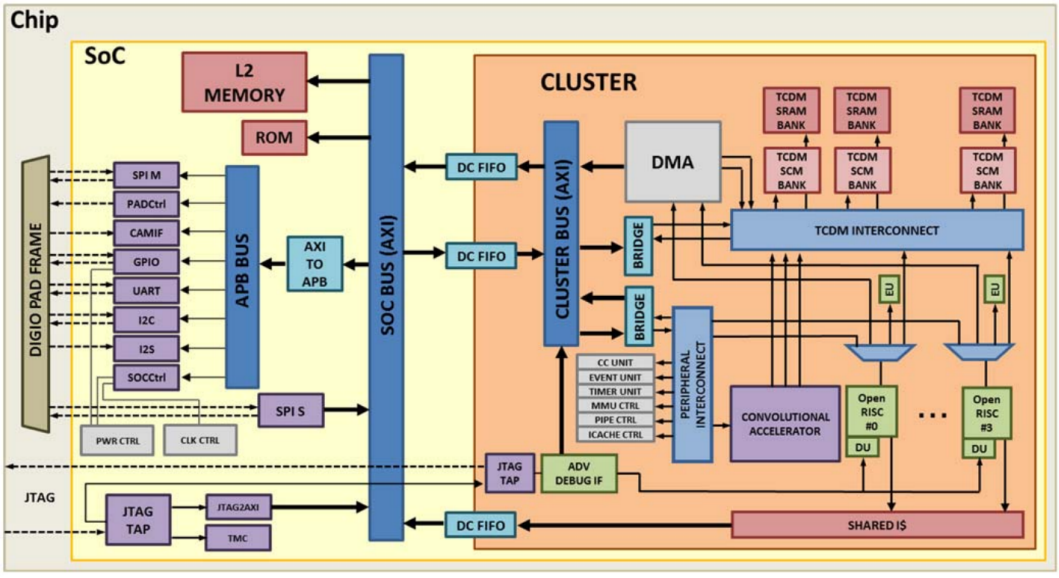


Figure 2.22: Architecture of Mia Wallace Source [25].

Figure 2.22 shows the Mia Wallace architecture design. As it can be seen, the hierarchical AMBA bus concept is used for the SoC communication design. Compared to the other designs, the implemented Joint Test Action Group (JTAG) standard is a unique feature. This provides better debugging ability for software developers. The clusters contain an internal AXI bus, a DMA controller, TCDM scratch-pad memory, OpenRISC OR10N cores and an CNN hardware accelerator, shared instruction cache, and two internal interconnects.

Mr Wolf [26]

The Mr Wolf design is a multi-core SoC design for edge IoT applications that has an advanced Input Output (IO) subsystem for highly efficient data acquisition for high amounts of sensor data. Mr Wolf provides full floating point support in a low-power design.

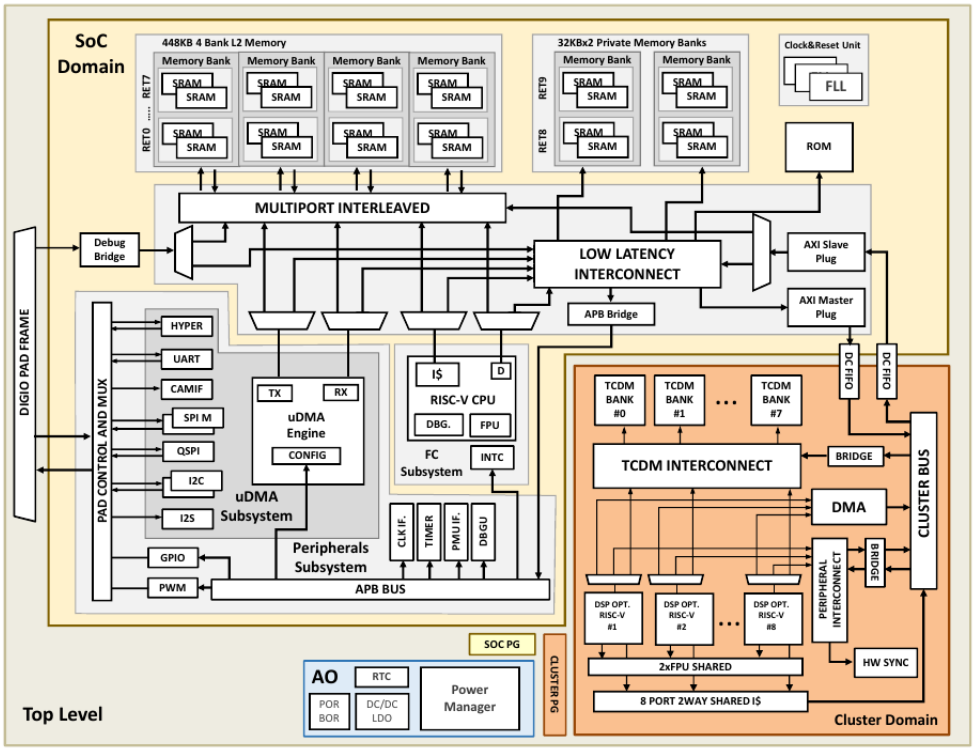


Figure 2.23: Architecture of the Mr Wolf design. Source [26].

Figure 2.23 shows the architecture of the Mr Wolf design. It can be seen that the architecture is split into a SoC domain and a cluster domain. The **SoC domain** has a peripheral subsystem for high bandwidth data acquisition, with its fabric controller for the peripheral control. There is a DMA engine to deal with the significant data amounts of the high bandwidth peripherals. Several private and shared memory banks are connected through a low latency interconnect on the SoC domain. The **cluster domain** contains several DSP-optimised RISC-V cores, TCDM memory banks, shared Floating Point Unit (FPU), DMA-controller, and several interconnects.

Snitch [27]

The Snitch system is a general-purpose computing system optimised for high energy efficient Floating Point (FP) arithmetic operations. Therefore, it uses Instruction Set Architecture (ISA) extensions to optimise the read and write from memory and optimise the floating points utilisation.

In Figure 2.24, the architecture of the Snitch can be observed. The system contains clusters connected through a AXI system crossbar. The **clusters** have their own AXI crossbar, which connects the cluster peripherals and hives. This cluster crossbar is also connected to the other clusters through the system crossbar. A **hive** is built of several snitch core complexes, TCDM scratch-pad memory. One shared instruction cache and a shared Multiplication/Division unit. Hives use TCDM scratch-pad memories instead of caches for data pre-loading and are connected to them through an interconnect. **Snitch core complexes** are made of a common RISC-V RV32G integer core and a FPU subsystem. The RV32G is a collection of RISC-V Standard extensions, and the G stands for general.

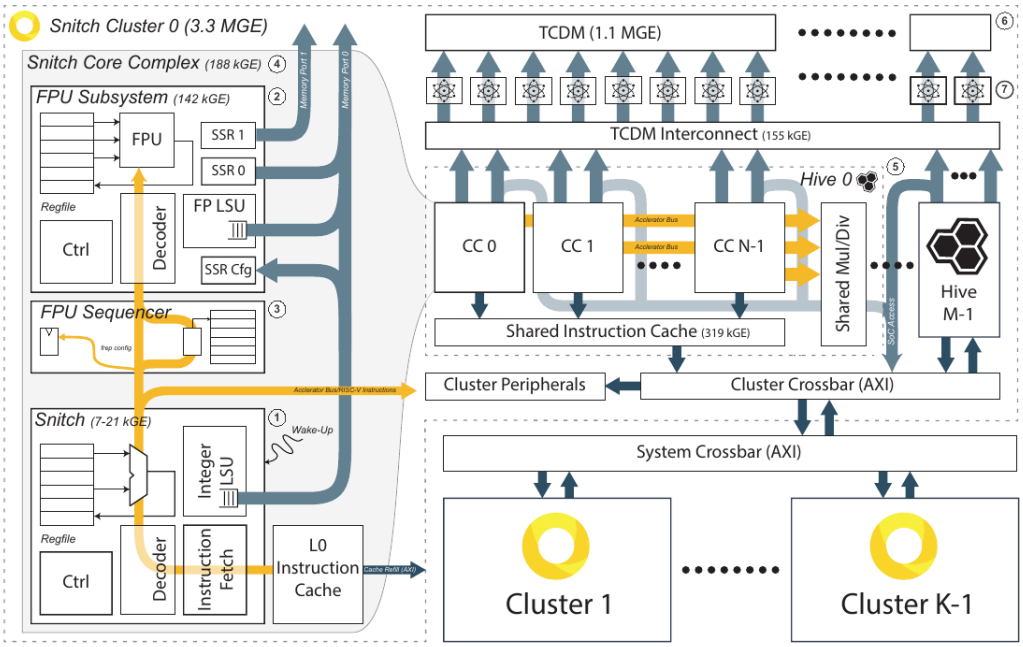


Figure 2.24: Architecture of Snitch. Source [27].

Overview of related embedded concepts

In Table 2.3, a comparison of the explained embedded multi-core systems can be seen. It can be observed that the use cases are quite different, except for Fulmine and Mia Wallace. This leads to different architectural designs. The only similarity is the usage of the AMBA bus system and the usage of TCDM scratch-pads (SP) instead of the L1 data cache. All of them are using DMA-controller, except the Snitch design. The designs are highly specialised to fulfil the requirements of a use case in a very efficient way.

	Fulmine	Mia Wallace	Mr Wolf	Snitch
Bus System	AMBA	AMBA	AMBA	AMBA
Use case	Encryption, CNN	CNN	Data acquisition	Efficient FP
Instruction cache	x	x	x	x
Cores Type	OR10N	OR10N	RCV32IMFX	RV32G
L1 Memory	TCDM SP	TCDM SP	TCDM SP	TCDM SP
FPU			x	x
Multi-core class	SMP	SMP	SMP	SMP
DMA controller	x	x	x	

Table 2.3: Comparison of existing embedded multi-core systems.

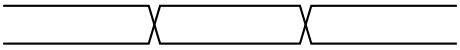
Conclusion of related embedded concepts

All related embedded multi-core architectures were designed to satisfy the requirements of a particular purpose. The designs were either trimmed to support hardware acceleration for special purposes or to efficiently handle significant amounts of data. All designs use the AMBA bus system in different topologies and caches. For this master thesis, real-time is an important topic, and caches are hard to use in real-time conditions. Therefore, no design is used in this master thesis. However, the gathered knowledge was used to design the own multi-core system.

CHAPTER 3

Implemented Work

This chapter explains the implemented hardware multi-core system and the created multi-core version of SmartOS. The hardware was developed in System Verilog and designed for easy re-usability and adoption. The OS is mainly written in C, and Assembler was used if necessary. Assembler code is needed to access the hardware register and initialise all data structures. A functional prototype was created and tested on the FPGA Artix-7-Nexys-4 [3].



3.1 Implemented hardware architecture overview

The implemented multi-core system has four cores and is connected by a WB Crossbar with an integrated bus arbiter. The WB bus system is used because of the user-defined tags for the bus signals. These signals are used for the bus arbiter. The current limitation for the number of cores is the used FPGA, as later shown in Section 4.1.1. The used FPGA would support more cores, as more resources on the FPGA are available. However, four cores are enough to test and demonstrate the hardware/software co-design. As the implemented work is easy to expand, more cores could be used in future work.

A schematic of the implemented multi-core system is shown in Figure 3.1. To handle all bus operations, the cores are connected through WB master controller modules and the slaves (Memory-, GPIO-, and UART modules) through slave interface modules. The WB master controller module creates the WB bus signals and is responsible for the different bus modes. The system currently supports the RMW mode, which is necessary for the correct IPC synchronisation.

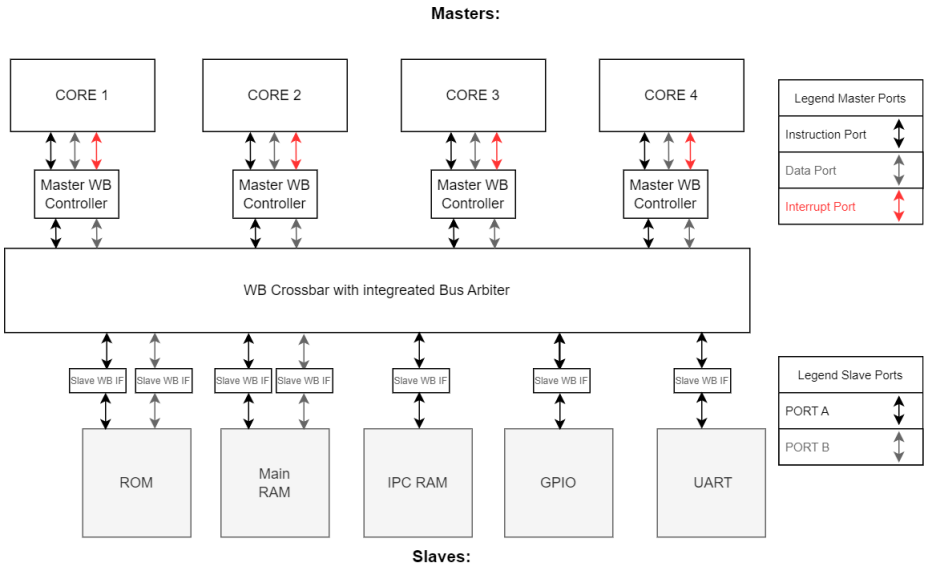


Figure 3.1: Schematic of implemented multi-core system.

A dual port ROM module was used for the nonvolatile memory. It has to be mentioned that for prototyping on the FPGA, the ROM synthesised as Block RAM (BRAM) cells and not real nonvolatile memory. This was done for easier prototyping of the multi-core system, as the system is only for demonstrating functionality.

The system has two RAM modules. One is a dual port RAM for the stack and heap for the OS of the cores, and the other is a single port RAM for IPC objects. The IPC RAM has one port to guarantee mutual exclusion access on the IPC objects.

The FPGA has a GPIO module and one UART module to provide a door to the outside of the multi-core system. These modules are mainly used for simulation, measurement and demonstration of the functionality of the created multi-core system. Therefore, for an extra core, only an additional core module with a corresponding master WB module has to be added.

Listing 3.1 shows the dynamic configurable implemented multi-core system, that is connected via dynamic interface modules. The number of slaves or masters can be changed via the local parameter, and the crossbar can handle dynamic numbers of slaves and masters.

```
localparam wb_slaves_num = 7;  // number of slaves
localparam wb_masters_num = 8; // two per core (data and instruction)

/* dynamic master interfaces generation */
core_if        core[wb_masters_num]();
wb_if_master   wbm[wb_masters_num]();

/*dynamic master interface generation */
slave_if       slaves[wb_slaves_num]();
wb_if_slave    wbs[wb_slaves_num]();

slave2wb slave0(.clk(clk),.rst(~rstn),.slave(slaves[0]),.wb(wbs[0]));
...
slave2wb slave6(.clk(clk),.rst(~rstn),.slave(slaves[6]),.wb(wbs[6]));

/* generation of the crossbar + integrated arbiter */
wb_crossbar_changeable_arbiter
#(
   //dynamic settings for number of slaves and master
   .MASTERCOUNT   (  wb_masters_num ),
   .SLAVECOUNT    (  wb_slaves_num  )
)
wishbone_arbiter_i
(
   .clk       (  clk   ),
   .rst       (  ~rstn ),
   .wb_masters(  wbm   ),
   .wb_slaves (  wbs   )
);
```

Listing 3.1: This code shows the dynamic configurable master and slaves interfaces.

3.1.1 Wishbone crossbar and integrated bus arbiter

Figure 3.2 is a schematic of the developed WB crossbar module that can be seen. The arbiter module is created as an integrated exchangeable System Verilog module to provide better adaptability for the system. This modular approach makes it possible to change the bus topology, reuse the arbiter module, or change the arbiter algorithm. To make this change as simple as possible, master and slave WB System Verilog interfaces are used.

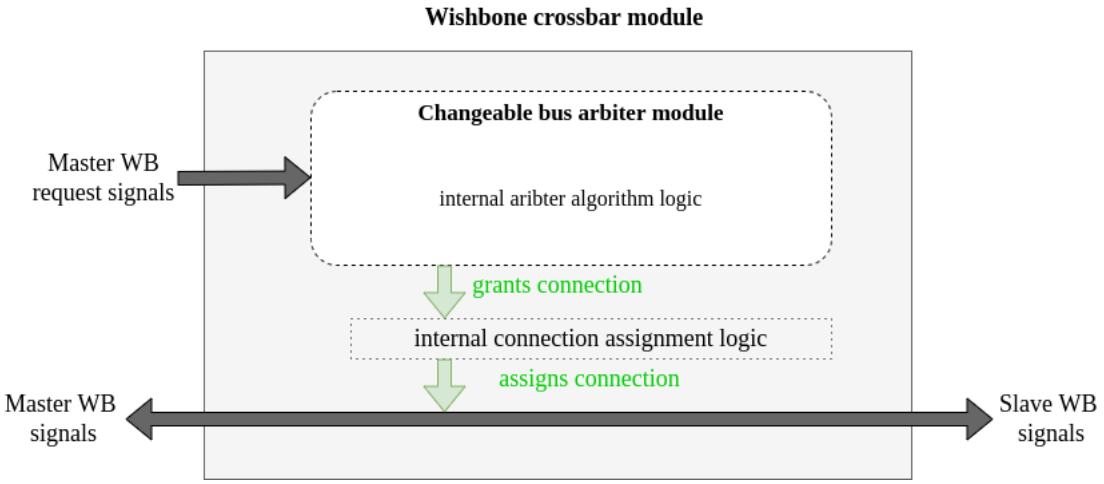


Figure 3.2: Schematic WB crossbar with integrated arbiter module.

As seen in Listing 3.1 line 23, the WB crossbar supports a configurable number of slaves and masters. The only limitation for the amount of connections will be resources on the FPGA. The crossbar has two primary interfaces, the slave (wbs) and master (wbm) port interfaces. This interface contains all WB signals needed for a successful connection.

Figure 3.2 shows a schematic of how a request is handled. The WB crossbar gets the WB request through the Master WB controller. This request is the high on the cyc signal of the master. Then, the bus arbiter assigns the request with the correct address and highest priority to an active connection. Requests with lower priorities must wait until the connection is free or until the other pending requests have a lower priority. The connection is terminated after the master WB controller gets an acknowledgement or error signal. The WB cyc signal is set to zero to signal the end of the connection to the arbiter. The slaves slavetoWB interface driver handles the WB signal and sends the acknowledge signal if the requested operation (read, write) is done; see Section 2.1.4.

The chosen arbiter algorithm for the multi-core system is the static priority scheme [7]. It uses the priority of the task of the cores for the algorithm scheme. The values for the priority are 0 to 255, whereas the highest value has the highest priority.

```
1  /* dynamic priority bus arbiter algorithm */
2  always_comb
3  begin
4     //set assigned masters to zero
5     for(int master_cnt = 0;master_cnt < MASTERCOUNT;master_cnt++)
6     begin
7     assigned_master[master_cnt] = 1'b0;
8     end
9     /* arbitation scheme grant next connection */
10    for(int slave_cnt = 0; slave_cnt < SLAVECOUNT; slave_cnt++)
11    begin
12       //default values for temp variables
13       temp_prio[slave_cnt] = 9'd256;
14       temp_winner[slave_cnt] = MASTERCOUNT;
15       for(int master_cnt = 0; master_cnt < MASTERCOUNT; master_cnt++)
16       begin
17          //default value for next connection
18          next_connection[master_cnt][slave_cnt] = 1'b0;
19          //assign next temp connection
20          if(pending_request[master_cnt][slave_cnt] &&
21             master_cyc_i[master_cnt] &&
22             ((temp_prio[slave_cnt] < {1'b0,master_tga_i[master_cnt]})
                    ||
23             temp_prio[slave_cnt] == 256)  && !assigned_master[
                  master_cnt])
24          begin
25             temp_prio[slave_cnt] = {1'b0,master_tga_i[master_cnt]};
26             temp_winner[slave_cnt] = master_cnt;
27          end
28       end
29       //all connections are checked assign next
30       if(temp_winner[slave_cnt] != MASTERCOUNT &&
31       !assigned_master[temp_winner[slave_cnt]])
32       begin
33           next_connection[temp_winner[slave_cnt]][slave_cnt] = 1'b1;
34           assigned_master[temp_winner[slave_cnt]] = 1'b1;
35       end
36    end
37 end
```

Listing 3.2: System Verilog code for the dynamic priority bus algorithm scheme.

The code for the algorithm can be seen in Listing 3.2. The algorithm starts at line 13 by assigning the temp variables values, which is impossible. As the counting starts with zero, the **MASTERCOUNT** value is not possible. The **temp_prio** 256 is higher than the highest possible task priority of the SmartOS, which is 255.

If a pending_request from a master to a slave is active, the request with the highest priority wins the arbitration. The code shows that the arbiter algorithm supports various slaves and masters. If two requests with the same priority coincide, the request with the lower master number wins.

3.1.2 Master Wishbone controller

The WB master controller converts the RISC-V data interface signals of the cores to wishbone signals. The controller is responsible for the correct bus mode. An atomic transfer bus mode is started if the controller detects an atomic instruction. In Figure 3.3, the processes of the master WB controller are shown. To make adaptability as easy as possible, all connections have defined interfaces to avoid handling too many signals in hardware design. The controller handles data and instructions signals and converts both to WB signals.

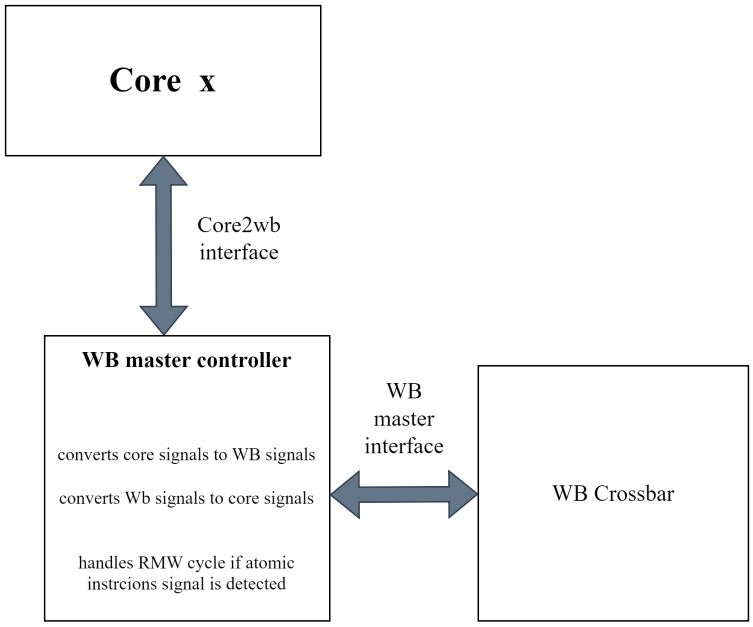


Figure 3.3: Schematic of master WB controller process.

The atomic transfer mode detection works through an atomic instruction signal that holds the part's opcode of the instruction. If an atomic load is detected, the atomic transfer bus mode is held until the next write operation is handled. Therefore, all atomic load instructions in the software design have to write back the loaded value. This is necessary to avoid too long blocking of the crossbar connection, otherwise the bus lock will be released with the next bus operation. See the System Verilog code in Listing 6.4 for implementation details.

3.1.3 Wishbone slave modules

The **SlaveToWb** interface driver handles the communication between slaves and the WB signals. The used peripheral or slave modules (RAM, ROM, GPIO and UART modules) were taken from the moreMCU project [28] and changed to support a WB connection. These changes mainly focused on timing issues with the WB bus signals.

3.1.4 Core modifications

The bus arbiter uses the priority of the running tasks on the core to control the bus requests. To get the priority of a task to the bus system, the first step is to save the task priority in a hardware register. This is where RISC-V's Control and Status Registers (CSRs) enter the game. The space from 0x800 to 0x8ff is not used in this register and can be used for a user-defined hardware register. Therefore, the register with the address 0x800 was created to save the task priority. In the Listing 3.3, an assembler snippet is shown; it can be seen how writing to this register is done.

```
1 // set task prio to 255
2 li  t0,0xFF
3 csrw 0x800, t0 // write to task priority register
```

Listing 3.3: Writing the task priority to a Control and Status Register (CSR).

The CSR 0x800 is routed through the core to the RISC-V data interface signals. Therefore, the bus arbiter can use the priority of the core's running task. The scheduler writes on ever context switch the priority of the task to the register.

3.2 Implemented operating system

The implemented multi-core SmartOS version is based on the moreMCU SmartOS single-core version. Figure 3.4 shows an overview of the multi-core concept. Every core runs its own OS on shared RAM with its data structures, and for the synchronisation between the cores, all use IPC objects. Nevertheless, the cores use the same OS code, except for the startup code. In the startup, the cores initialise their OS data structures and set their priority for the bus system.

The OS of core 1 operates as a master, as it is responsible for the initialisation of the IPC object in its startup code. Therefore, core 1 starts with the highest possible priority after a reset. All other cores start with the second highest priority to ensure core 1 wins all bus access races. This design choice ensures that all IPC objects are correctly set up for their first usage. An example and a better explanation for this priority process can be seen in Section 4.2.1.

In Figure 3.4, a schematic of the implemented OS is shown. Every core has its own OS and uses a shared ROM and RAM.

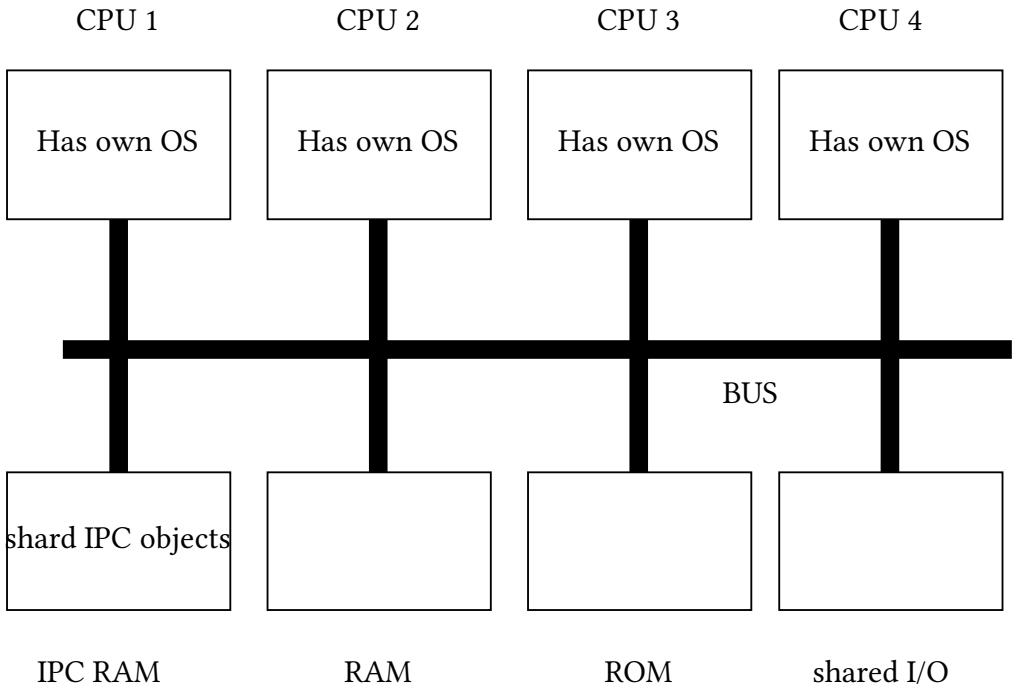


Figure 3.4: Schematic of implemented multi-core OS.

3.2.1 Memory Operating System concepts

The implemented multi-core system has three memory modules, the ROM, the RAM and IPC RAM. The IPC RAM is a single port memory. All IPC objects are saved there. It has only one port to simplify the atomic bus mode. All objects saved in the IPC RAM are aligned to the 32-bit word size or a multiple. The linker with the align function saves IPC to avert unaligned positioning in memory. However, the linker, does ignore the align commands because of bugs. Mutex lock operations failed because of two separate load instructions caused by unaligned positioning. Therefore, all lock variables were increased to word size, and the message structure uses padding to avoid unaligned memory positioning.

Figure 3.5 shows an overview of the used memory concept. The ROM can be grouped into shared and core-specific regions. Every core has its regions for its variables and constructors. In the core-specific entry or startup code, these data variables are initialised and put into the RAM. The startup code also initialises the IPC object. Figure 3.5 does not have the correct order of how the sections are positioned in memory, as the schematic is an overview. The RAM layout has a separate 12 Kibibyte (KiB) memory region for every core.

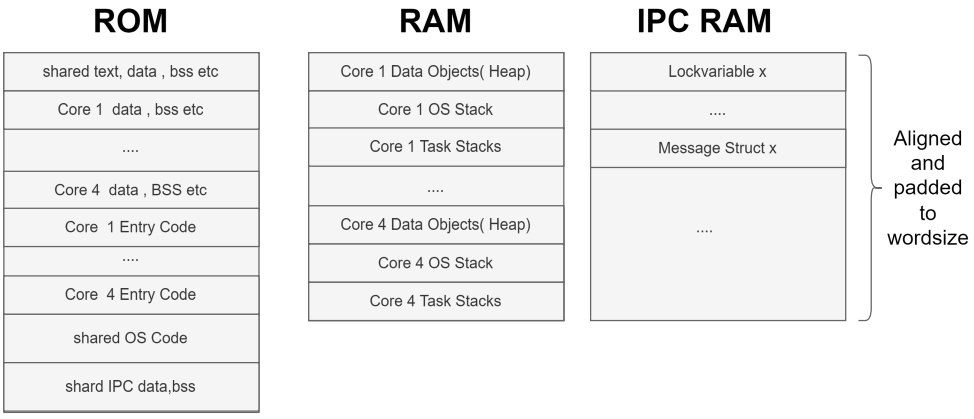


Figure 3.5: Schematic of memory OS concept.

3.2.2 Shared core Operating System concepts

In Listing 3.4, examples of the different macros for the OS data objects can be seen. The first macro is for task creation of core 3. The most important part of the macro is the compiler attribute SECTION; this ensures that the task is put into the right ROM memory position. The right ROM memory position is the constructor table for the corresponding core.

```
1  /** Creates a task for core 3 */
2  #define OS_TASK_CONSTR_CORE3(_name, _stack_size, _priority)  \
3      void entry_##_name(void);                \
4      TaskConstr_t __taskConstr_##_name __SECTION(".os_taskConstr_core3
           ") = \
5        {                                \
6           .name = #_name,               \
7           .entry = &entry_##_name,      \
8           .prio = _priority,            \
9           .stackSize = _stack_size      \
10     }
11
12 /** Creates a message_mailbox_struct */
13 #define OS_CREATE_MESSAGE_MAILBOX_STRUCT(_name)  \
14       Mailbox_t message_mailbox_struct_##_name __SECTION(".
             os_IPC_data") = {0}
15 /** Creates a mutex */
16 #define OS_CREATE_MUTEX(_name) \
17       Lock_var_t mutex_##_name __SECTION(".os_IPC_data") = {0}
18 /** Creates a spinlock*/
19 #define OS_CREATE_SPINLOCK(_name)  \
20       Lock_var_t spinlock_##_name __SECTION(".os_IPC_data") = {0}
```

Listing 3.4: Predefined macros for OS task - and IPC objects.

All tasks have their own ROM memory region. Therefore, every core has different start and end positions of the **os_task_ccb** objects. This makes it possible for every core to iterate over its task constructor object and just create tasks for its core. In Listing 3.4, an example for tasks of core 3 is shown. However, the macro works for all cores the same, and only the target memory position differs.

The other shown macros of Listing 3.4 are for creating IPC objects. IPC are mapped to a particular memory region and later put into the IPC RAM.

```
1 volatile uint8_t core;
2 __asm__ volatile("csrr %0, 0xF14":"=r"(core));
3 core &= 0xF;
4 switch (core)
5 {
6    case 1:
7    {
8       // core specifc operations
9    }
10   ...
11   case 4:
12   {
13   }
14
15 }
```

Listing 3.5: Difference of shared code between Cores.

Listing 3.5 shows an example of how the shared code is specified to a defined core is shown. The key concept is the extension of the core of a particular register, which holds the core and cluster numbers. These hardware registers are set in the system verilog code for the multi-core system. Therefore, the core number can be read with the assembler instruction in line 2. This makes it possible for every core to run the code that belongs to it.

3.2.3 Spinlock implementation

Spinlocks are only used in the current OS implementation during the operating system's startup. At this moment in the concept of SmartOS, no sleep or yield function could be used. The operating system data structures are not initialised, and no tasks are running, only the start-up code. Therefore, the spinlocks are used to synchronise the printfx operations. In Listing 3.6, the lock function of spinlock can be seen. As the lock variable **lockvar_printfx** is used, the functions provide no interface for a dynamic lock variable during the startup.

```
1 void __attribute__((optimize("-O0")))spinlock_lock_printfx()
2 {
3     //load lock variable
4     /*** start atomic core section ***/
5     DISABLE_INTERRUPTS // Marco
6     __asm__("lui   t0, %hi(lockvar_printfx)");
7     __asm__("addi  t0,t0,%lo(lockvar_printfx)");
8     __asm__("trylockspin:");
9     __asm__( "li t1,1");
10    //check value of lock variable
11    /****** start atomic bus section ******/
12    __asm__("lr.w.aq t2,(t0)");
13    //jump to retry if not zero
14    __asm__( "bnez t2,retry");
15    __asm__( "sc.w.aq t3,t1,(t0)");
16    __asm__( "j end");
17
18    __asm__("retry:");
19    //write a 1 to lock var to break atomic transferbusmode
20    __asm__( "sc.w.aq t3,t1,(t0)");
21    /****** End atomic bus section ******/
22    ACTIVATE_INTERRUPTS // Marco
23    /*** End atomic core section ***/
24    //break atomic section with some nop to avoid deadlock
25    for(int i = 0; i < 15; i++)
26      ;;
27    __asm__("j trylockspin");
28    __asm__("end:");
29
30 }
```

Listing 3.6: Implemented spinlock lock function of the OS.

In the lock function, first, the interrupts for the core are disabled with the macro **DISABLE_INTERRUPTS**. This macro starts the atomic section for the core. The value of the lock variable is loaded with the atomic load reserved (lr.w.aq). This starts an atomic transfer mode in the WB bus. To end the mode, a write operation has to be done. That makes it necessary to write the value one back to the lock variable. It does not affect the lock variable but ends the atomic transfer mode and ends the atomic section for the bus.

This design issue can decrease the performance of the bus if the bus is locked too long or is not unlocked. Therefore, after an atomic load reserved instruction, an atomic store conditional is mandatory.

Before the retry part, the macro **ENABLE_INTERRUPTS** ends the atomic section on the core. In the retry part, the spinlock performs 15 nop instructions to give other cores the time

to unlock the spinlock. This is done to avoid deadlocks, where the highest priority core blocks a lower priority core from unlocking the spinlock.

In Listing 3.7, the unlocking function of the spinlock can be seen. It uses an atomic instruction to set the lock variable to zero. This operation does not start an atomic transfer mode cycle on the bus.

```
1 void  spinlock_unlock_printfx()
2 {
3   __asm__("lui   t0, %hi(lockvar_printfx)");
4   __asm__("addi  t0,t0,%lo(lockvar_printfx)");
5   __asm__("sc.w.rl zero,zero,(t0)");
6
7 }
```

Listing 3.7: Implemented spinlock unlock function of the OS.

3.2.4 Mutex implementation

The busy free waiting IPC concept is the mutex. In the OS, the mutex is used as soon as the tasks are ready. A mutex is the non-blocking IPC concept of the OS and uses the yield or sleep function to let other tasks use computational time. As seen in Section 3.2.3, the mutex must also operate a write instruction in the retry part to end the atomic transfer bus mode. In contrast to the spinlock functions, the mutex functions have an interface for different mutex objects. In opposition to the spinlock, the mutex is used for mutual exclusion after the OS is initialised.

```
1  void __attribute__((optimize("-O0"))) mutex_lock(Lock_var_t * mutex)
2  {
3     // no load of mutex address needed already in a0
4     __asm__("mutexaq:");
5     __asm__( "li t1,1");
6     //atomic load of lock variable value
7     __asm__("lr.w.aq t2,(a0)");
8     __asm__( "bnez t2,retry");
9     __asm__( "sc.w.aq t3,t1,(a0)");
10    __asm__( "j end");
11
12    __asm__("retry:");
13    __asm__( "sc.w.aq t3,t1,(a0)");
14    //write a 1 to lock var to break atomic transferbusmode
15    yield();
16    __asm__("j mutexaq");
17    __asm__("end:");
18 }
19 void __attribute__((optimize("-O0"))) mutex_unlock(Lock_var_t* mutex)
20 {
21    // no load of mutex address needed already in a0
22    //write zero to lock variable
23    __asm__("sc.w.rl zero,zero,(a0)");
24 }
```

Listing 3.8: Implemented mutex of the OS.

The mutex's unlock function does not differ from the spinlock implementation and is only shown for completeness. It only differs in the implementation as it has a dynamic interface for different mutex lock variables. The concept is only real-time capable if it is used to fulfil all deadlines. Therefore, the code has to be analysed, and the lock and unlock functions must be timed to reach the deadlines.

3.2.5 Message-passing implementation

A message-passing concept was implemented to exchange information between tasks on the same core or a different core. The concept uses a mailbox data structure to share information between tasks. The implementation supports different message-passing designs. The send and receive processes can be non-blocking or blocking. The mailbox system supports a one-to-one, one-to-many, or many-to-many approach.

The basis of the concept is a message mailbox structure. This data structure contains a lock variable and a message queue. The mailbox comprises a destination ID, source ID and data buffer. The currently implemented mailbox can store four messages, and 8 Bytes of data can be exchanged. A mutex secures all manipulation operations to ensure mutual exclusion access to the message data. Listings 3.9 and 3.10 show the code for the send and receive functions for the message-passing concept.

```
1 Result_t  send(Mailbox_t * message_mailbox_struct,uint8_t dest_id,
      uint8_t source_id,char * data)
2 {
3    mutex_lock(&message_mailbox_struct->mutex);
4    uint8_t i = 0;
5    //go trough mailbox
6    while( i < 4)
7    {
8       //dest_id is zero message place is empty
9       if(message_mailbox_struct->destination_id[i] == 0)
10      {
11         message_mailbox_struct->source_id[i] = source_id;
12         message_mailbox_struct->destination_id[i] = dest_id;
13         memcpy(message_mailbox_struct->message_cont[i],data,8);
14         mutex_unlock(&message_mailbox_struct->mutex);
15
16         return SUCCESS;
17      }
18      i++;
19   }
20   mutex_unlock(&message_mailbox_struct->mutex);
21   //mailbox is full
22   return FAILURE;
23 }
```

Listing 3.9: Send function for message passing of the OS.

As seen in Listing 3.9, the send function interface holds the message mailbox data struct, a destination ID, a source ID and the corresponding send data. The source ID makes the many-to-x design possible, and the destination ID makes the x-to-many design possible.

Both message-passing functions have a return value indicating whether the mailbox is empty or finished, depending on the caller (Producer, Consumer). This design approach enables handling send or receive operations non-blocking or blocking. As an example, the message queue of the mailbox is full, and the send function fails as it can not add the message to the mailbox. The developer can decide whether to block the call until the message can be added to the message queue or continue the task's business.

As can be seen in the send functions, the mailbox is checked to see if it has free space for a new message. A message space is empty if the destination ID is zero. The first initialisation of the mailbox and the destination IDs is done in the startup code of core 1. If an empty space is found, the message is placed in it, and SUCCESS is returned.

In Listing 3.10, the receive function of the message-passing implementation can be seen. The mailbox is checked to see if it contains a message for the consumer. If the message is for the consumer, it is consumed by setting its destination ID to zero. If a message for the consumer was found in the mailbox, SUCCESS is returned. Otherwise, the receive function shows that there was no message with FAILURE.

```
1 Result_t  receive(Mailbox_t * message_mailbox_struct,uint8_t dest_id,
      uint8_t * source_id,char * data)
2 {
3    mutex_lock(&message_mailbox_struct->mutex);
4    uint8_t i = 0;
5    //go trough mailbox
6    while( i < 4)
7    {
8       if(message_mailbox_struct->destination_id[i] == dest_id)
9       {
10          *source_id = message_mailbox_struct->source_id[i];
11          //clear destination_id to show message was consumed
12          message_mailbox_struct->destination_id[i] = 0;
13          memcpy(data,message_mailbox_struct->message_cont[i],8);
14          mutex_unlock(&message_mailbox_struct->mutex);
15          return SUCCESS;
16       }
17       i++;
18    }
19    mutex_unlock(&message_mailbox_struct->mutex);
20    return FAILURE;
21 }
```

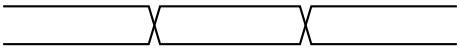
Listing 3.10: Receive function for message passing of the OS.

This section shows that the implemented message-passing system is highly flexible and can be used for many use cases. This design should make exchanging data on the new SmartOS multi-core OS simple.

CHAPTER 4

Measurement and Simulation

This chapter examines the implemented multi-core system and OS. The first part consists of hardware results, where the resource consumption of the implemented multi-core system is shown. The implemented four-core multi-system and an expanded six-core system are compared. The second part of this Chapter is about measurements and simulations of the hardware/software co-design. The implemented crossbar is shown, and the IPC concepts are measured and shown by a proof of concept.



4.1 Hardware results

4.1.1 Resource consumption

This section shows the resource usage for 4-core multi-core and 6-core-multi-core systems. All resources that the FPGA consists of are listed in Section 4.1.1 and explained. The multi-core systems were synthesised with the Vivado toolchain. This software provides a utilisation report that shows how many resources of the FPGA Artix-7-Nexys-4 are used. The hardware was developed with the approach of easy expandability. For this reason, an analysis for scalability is made for the developed modules in this section.

FPGA resources short overview

The components of the Artix-7-Nexys4 FPGA are Look-up Table (LUT), BRAM, Flip Flop (FF), Mixed-Mode Clock Manager (MMCM) and Digital Signal Processing (DSP) and IO. LUTRAM. The LUT is used to implement the logic function. The BRAM configurable memory block stores large data. The FF is to keep data on every clock cycle, latch data, or enable signals. MMCM are used to generate clock signals. DSP that provides arithmetic pre-adder and multiplier features. LUTRAM can be used for logic function and memory.

Resource usage of a 4-core multi-core system

The subsequent tables show the resource usage of the implemented four-core multi-core system. A detailed overview of the architecture of the multi-core system can be seen in Section 3.1. The multi-core system has 48 KiB of ROM and RAM, 1KiB of IPC RAM. This size is more than enough for code and task data. All cores are connected to the WB master controller to the crossbar. The system has a GPIO module and UART module connected to the crossbar via the **slave2wb** interfaces. The memory modules are also connected to the crossbar via the **slave2wb** interface. The main RAM and ROM use both boards. Therefore, every main memory module has two **slave2wb** interfaces.

In Table 4.1, the component usage of the FPGA for the peripherals, one core and in total is listed. The design uses 44.46% of all LUTs and 8.39% of the FFs. As seen in the table, peripherals use 5.374% of all LUTs and only 0.2% of the FFs. The cause is that peripherals require fewer logic operations than the cores and have to handle fewer data signals. Cores do all calculations and handle all data signals for their internal components. Therefore, they need many LUTs for logic functions and many FFs for the data signal handling. The system uses 25% of the BRAM components. As seen in Table 4.1, the peripherals use all of the memory. However, memory is not helpful for the analysis. As the memory size was chosen to be bigger than needed, the ROM could save code for more cores. For the OS, every core gets a 12 KiB memory region. This calculation implies for a system with more cores, 12 KiB of RAM should be added. Only one MMCM component is used for the clock generator. Therefore, MMCM components can be neglected for the scalability metric. Five DSP are used per core for the Multiplier–accumulator (MAC) unit.

Resource	Utilisation peripherals	Overall-utilisation in % peripherals	Utilisation 1 core	Overall-utilisation in % 1 core	Utilisation all	Overall-utilisation in % all
LUT	3407	5.374	6194,5	9.771	28185	44.46
LUTRAM	16	0.08	0	0	16	0.08
FF	254	0.2	2595	2.0465	10634	8.39
BRAM	34	25.19	0	0	34	25.19
DSP	0	0	5	2.0825	20	8.33
IO	68	0	0	0	68	32.38
MMCM	1	16.67	0	0	1	16.67

Table 4.1: Used components of the FPGA Artix-7-Nexys-4 in percentage of a 4-core system.

Table 4.2 shows a detailed view of all used components. As it can be seen, the WB master controller and **slave2wb** interface use a low number of LUTs and registers. These modules must only convert the WB signals to core or slave signals and do not need much logic operations or data handling for their internal components. Thus, these modules could be neglected for the scalability analysis, as the low number does influence the scalability little. The used components per core stay almost constant. The used DSP per core is constantly 5 as every core only has one MAC unit. The number of used LUTs per core varies slightly, and so does the number of Multiplexer (MUXES). The optimisation of the synthesis causes this slight variation. Therefore, the average of all cores is used for the utilisation of 1 core in Table 4.1.

Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Slice (15850)	LUT as Logic (63400)	LUT as Memory (19000)	Block RAM Tile (135)	DSPs (240)	BUFGCTRL (32)	MMCME_ADV (6)
Total Used:	28185	10634	1643	4	8180	28169	16	34	20	2	1
Clock generator	0	0	0	0	0	0	0	0	0	2	1
WB master controller 1	6	3	0	0	5	6	0	0	0	0	0
WB master controller 2	6	3	0	0	6	6	0	0	0	0	0
WB master controller 3	6	3	0	0	6	6	0	0	0	0	0
WB master controller 4	6	3	0	0	6	6	0	0	0	0	0
GPIO module	24	33	0	0	34	24	0	0	0	0	0
RAM Cores	16	0	0	0	6	16	0	16	0	0	0
RAM IPC	8	0	0	0	4	8	0	2	0	0	0
Core 1	6212	2601	400	0	1899	6212	0	0	5	0	0
Core 2	6205	2593	402	2	1831	6205	0	0	5	0	0
Core 3	6150	2593	423	0	1852	6150	0	0	5	0	0
Core 4	6211	2593	403	2	1965	6211	0	0	5	0	0
ROM	6	0	0	0	4	6	0	16	0	0	0
Slave IF 0	2	1	0	0	2	2	0	0	0	0	0
Slave IF 1	2	1	0	0	1	2	0	0	0	0	0
Slave IF 2	1	1	0	0	1	1	0	0	0	0	0
Slave IF 3	1	1	0	0	1	1	0	0	0	0	0
Slave IF 4	1	1	0	0	2	1	0	0	0	0	0
Slave IF 5	2	1	0	0	3	2	0	0	0	0	0
Slave IF 6	1	1	0	0	2	1	0	0	0	0	0
UART module	157	146	0	0	87	141	16	0	0	0	0
WB Crossbar	3140	56	0	0	1044	3140	0	0	0	0	0

Table 4.2: All used components of the FPGA Artix-7-Nexys-4, for a 4-core system.

Table 4.2 shows LUTRAM is only used by the UART module. The UART module uses this memory for the internal transfer and receive buffer. The BUFGCTRL and MMCME_ADV resources are MMCM components used for the clock generation. As all implemented multi-core systems use one clock and one UART module, the number of used resources stays constant. Both outliers are not relevant to the scalability of the system as all clocks use the same clock generator, regardless of the number of cores. The number of resources used by the UART module is so small that it can be neglected.

The significant module for the scalability of the multi-core system is the crossbar. As for every newly added core or slave, new connections in the crossbar are needed. This increasing number of connections increases the resource consumption of the peripherals. The crossbar uses 3140 LUT, thus 11.14% of all used LUT and 4,95% of all available LUT of the FPGA. The LUT are mainly used for the logic function of the internal arbiter of the crossbar. The number of FF is only 56. Therefore, this does not make an impact in comparison to the overall used FF (10634).

Resource usage of a 6-core multi-core system

The following table shows the resource consumption for a 6-core multi-core system. This system was created to show how well the implemented modules scale. In Figure 4.3, it can be seen that the overall usage of LUT increased to 65.95%. Therefore, 21.49%, more LUT are used than in the four-core system. 19.728% LUT overall usage is caused by the 2 additional cores. The last 1.762% increase is caused by the increased complexity of the crossbar and the internal arbiter. The total FF components usage increased by 4.17%, from 8.33% to 12.5%. This increase can also explained by the the additional two core modules. The increased 0.027% of total FF components usage by the crossbar are so low that they can be neglected for scalability. The memory usage is not relevant for the scalability, as the ROM was increased to 88 KiB and the RAM to 192 KiB. To test the limits of the memory that the FPGA can support.

Resource	Utilisation peripherals	Overall-utilisation in % peripherals	Utilisation 1 core	Overall-utilisation in % 1 core	Utilisation all	Overall-utilisation in % all
LUT	4290	6.767	6254	9.864	41814	65.95
LUTRAM	16	0.08	0	0	16	0.08
FF	288	0.227	2593,66	2,045	15850	12.50
BRAM	98	72.59	0	0	98	72.59
DSP	0	0	5	2.0825	30	12.50
IO	68	0	0	0	68	32.38
MMCM	1	16.67	0	0	1	16.67

Table 4.3: Used components of the FPGA Artix-7-Nexys-4 in percentage of a 6-core system.

The DSP components raised as expected to 30 due to the 5 DSPs used per MAC unit. Also, as expected, the number of used LUT and FF per core stayed stable. Again, the slight variation caused by the synthesis optimisation can be seen. The outliers of MMCM and LUTRAM from before stayed the same, as no more UART module was added, and only one clock generator was used.

The interesting numbers are the components used in the crossbar. The number of used LUT increased from 3140 to 4210, a rise of 34%. However, in the overall LUT usage, this rise is only 1.762%. Two more cores increase the needed logic operations for the connection handling and the arbiter algorithm, and therefore, the rise of 34% LUT usage can be explained. But as the overall LUT rise is only 1.762%, it has a low effect on the overall scalability. All other peripheral resource usage shows a rise caused by the two more needed WB master controllers and increased memory for the cores.

Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Slice (15850)	LUT as Logic (63400)	LUT as Memory (19000)	Block RAM Tile (135)	DSPs (240)	BUFGCTRL (32)	MMCME_ADV (6)
Total Used:	41814	15850	2375	3	11795	41798	16	98	30	2	1
Clock generator	0	0	0	0	0	0	0	0	0	2	1
WB master controller 1	6	3	0	0	6	6	0	0	0	0	0
WB master controller 2	6	3	0	0	7	6	0	0	0	0	0
WB master controller 3	6	3	0	0	7	6	0	0	0	0	0
WB master controller 4	6	3	0	0	6	6	0	0	0	0	0
WB master controller 5	6	3	0	0	7	6	0	0	0	0	0
WB master controller 6	6	3	0	0	6	6	0	0	0	0	0
GPIO module	24	33	0	0	33	24	0	0	0	0	0
RAM Cores	43	0	0	0	32	43	0	64	0	0	0
RAM IPC	8	0	0	0	4	8	0	2	0	0	0
Core 1	6228	2597	385	0	2039	6228	0	0	5	0	0
Core 2	6226	2593	385	0	1879	6226	0	0	5	0	0
Core 3	6265	2593	384	0	1953	6265	0	0	5	0	0
Core 4	6198	2593	401	1	2089	6198	0	0	5	0	0
Core 5	6192	2593	401	1	1911	6192	0	0	5	0	0
Core 6	6195	2593	401	1	1856	6195	0	0	5	0	0
ROM	5	0	0	0	3	5	0	32	0	0	0
Slave IF 0	2	1	0	0	2	2	0	0	0	0	0
Slave IF 1	2	1	0	0	2	2	0	0	0	0	0
Slave IF 2	1	1	0	0	2	1	0	0	0	0	0
Slave IF 3	1	1	0	0	2	1	0	0	0	0	0
Slave IF 4	1	1	0	0	2	1	0	0	0	0	0
Slave IF 5	2	1	0	0	2	2	0	0	0	0	0
Slave IF 6	1	1	0	0	2	1	0	0	0	0	0
UART module	157	146	0	0	81	1	16	0	0	0	0
WB Crossbar	4210	84	3	0	1432	141	0	0	0	0	0

Table 4.4: All used components of the FPGA Artix-7-Nexys-4, for a 6-core system.

This section shows that the chosen design scales well for the used FPGA. The crossbar does not limit scalability for the used FPGA; therefore, it fits very well. The limitation is the core internal logic functions and therefore used LUT. If the numbers scale linear, 9 cores are the maximal number that can be used in a multi-core system on the used hardware.

4.2 Hardware/software co-design

This section shows the functionality implemented in hardware/software co-design via simulation and measurement on the FPGA. The simulation was made with Vivado 2020.1, more accurately with the simulation waveform tool of it. This tool was intensively used for developing part of the OS, as the hardware does not support debugging. For the measurement part of the hardware/software co-design, a Hantek two-channel digital oscilloscope [29] and CuteCom [30] were used.

4.2.1 Dynamic bus priorities of cores simulation

The WB bus systems arbiter chooses who is next on the priority of the currently running tasks of the cores. Every core has a separate data and instruction bus. Therefore, the priority of the data and instruction bus is shown in Figure 4.1. The WB user-defined address tag signal **tga** is used to transport the priority to the arbiter (see Section 3.1.1). This simulation was recorded to show how the initialisation problem of IPC data was solved. At a startup of the OS, the values of memory of the IPC data are not defined. Therefore, the IPC data must be initialised with the correct value in memory. This is done by core 1 in the initialisation phase of the OS (see Listing 6.5).

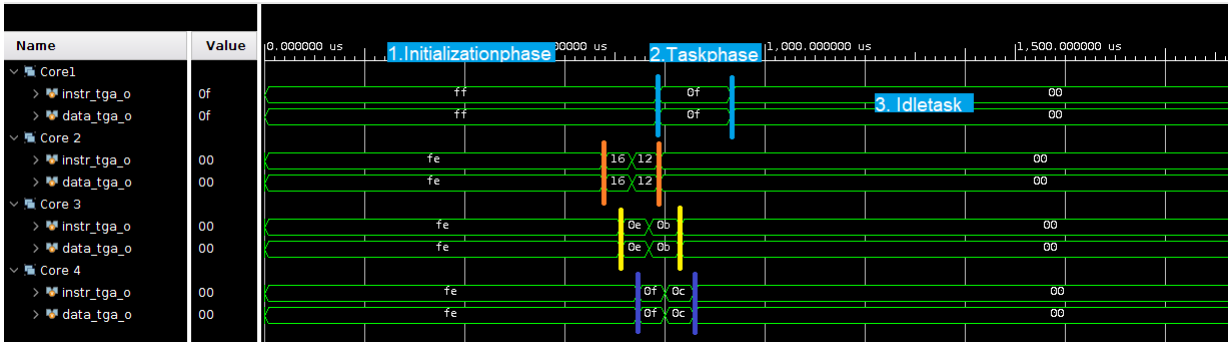


Figure 4.1: This shows the dynamic bus priorities of the cores, starting at a reset.

As can be seen in Figure 4.1, core 1 starts with the highest priority (0xFF) after a reset (see Marker 1). The current bus priority of a core can be seen in the **instr_tga_o** and **data_tga_o** signals. Figure 4.1 shows all four cores' instructions and data bus priorities. All other tasks start with a lower priority (0xFE). Therefore, core 1 always wins the race in a direct bus request battle. This high priority ensures that in the core 1 startup code, the IPC structures are initialised before the first usage.

In Figure 4.1, the priority of the cores changes after the first context switch and the first tasks are running (see Figure 4.1 Marker 2). In detail, for core 2, the priority of the first task is 0x14, and for the second task, 0x12. A priority of 0 shows that the idle task is running on a core (see Marker 3).

4.2.2 Wishbone crossbar with integrated arbiter

This section shows the implemented features of the crossbar with the integrated bus arbiter for the 4-core multi-core system. The access time of a single non-pipelined bus transfer is measured, and the atomic bus transfer mode is shown in the simulation.

4.2.3 Crossbar bus access time

Figure 4.2 shows several single non-pipelined bus transfers, and the focus is on the access time in clock cycles for one single non-pipelined bus transfer (see Figure 4.2 Marker 1) of the 4-core multi-core system. The bus requests of core 1, core 2, and core 4 are for the ROM (address range 0x0 to 0x00400000); this can be seen at the signals **adr_o**. Therefore, all bus requests are in a race to the bus access.

The simulation was performed in the initialisation of OS. This can be seen by the high priorities of the **tga_o** signals (see Section 4.2.1). The bus transfer of core 1 starts with the high signal of the cycle signals **cyc_o** (see Figure 4.2 Marker 2). This cycle-high signal is made by the WB master controller, which indicates to the arbiter that a bus request is pending.

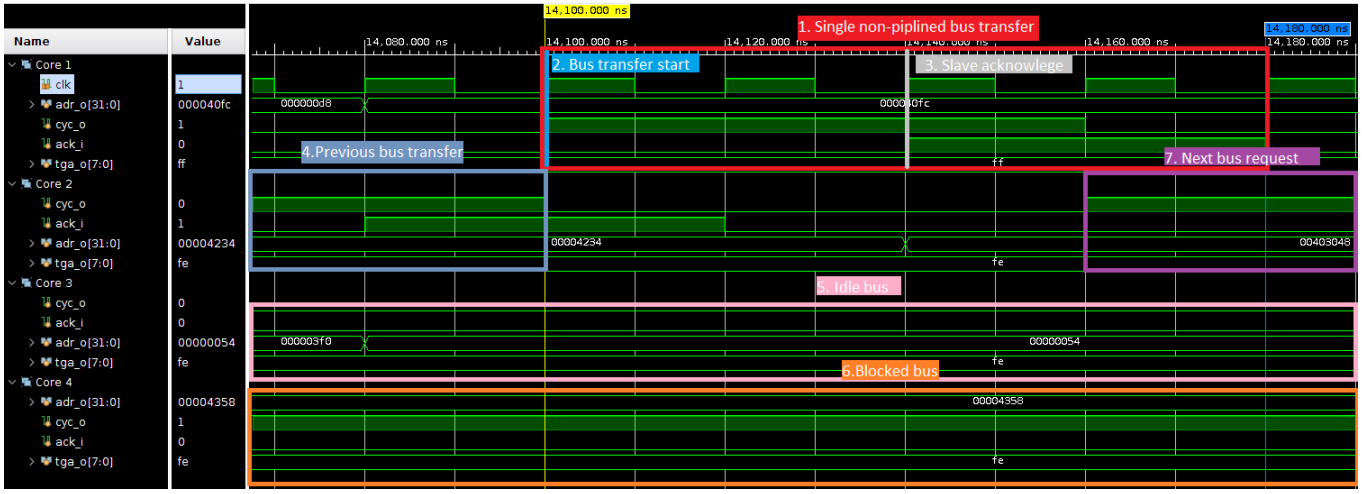


Figure 4.2: This shows the access time of core one single non-pipelined bus transfer, shown in the red rectangle (Marker 1).

The integrated bus arbiter of the crossbar needs an additional clock cycle to grant a connection. The bus request of core 1 has the highest priority (0xFF) and, therefore, blocks the lower priority bus request of core 4 (see Figure 4.2 Marker 6). Thus, the bus arbiter instantly grants the connection. The slave acknowledges the signal after another clock cycle. This additional clock cycle is due to read or write operations taking one or more clock cycles on the slave side. After the master receives the acknowledge signal, the master controller indicates the end of the connection by setting the cycle signal to low. The crossbar ends the

connection on the next subsequent clock signal, as seen in the high acknowledge signal of the connection.

A single non-pipelined bus transfer's best-case access time can be seen, and it is 4 clock cycles long. This best-case scenario is only possible if a bus request has the highest priority. In Figure 4.2, it can be seen that core 2 and 4 bus connections to the ROM (address 0x0 to 0x400000) are blocked by higher priority requests. Therefore, core 4 has to wait until no higher priority bus request uses the connection. Core 4 has the same priority as core 2; in this case, the core with the lower core number wins the race (see Section 3.1.1).

The worst-case bus access time depends on the priority of the bus requests and the number of bus requests; therefore, a low-priority task can be blocked for a certain number of clock cycles. In all testing of the multi-core system, no core bus requests were blocked forever till low priority tasks. This can be explained by the fact that a core does not use all its clock cycles for bus requests, and a crossbar has separate connections to the slaves. Also, the bus transfers are not preemptive, and therefore, low-priority bus accesses can not be interrupted.

4.2.4 Atomic bus transfer mode

The atomic bus transfer can be seen in Figure 4.3. This bus mode is essential for the IPC concepts. The synchronisation would fail without this mode as the read-modify write of lock variables would not be mutually exclusive. Therefore, any other master can interrupt the access of a lock variable and corrupt it (see Section 2.2.3).

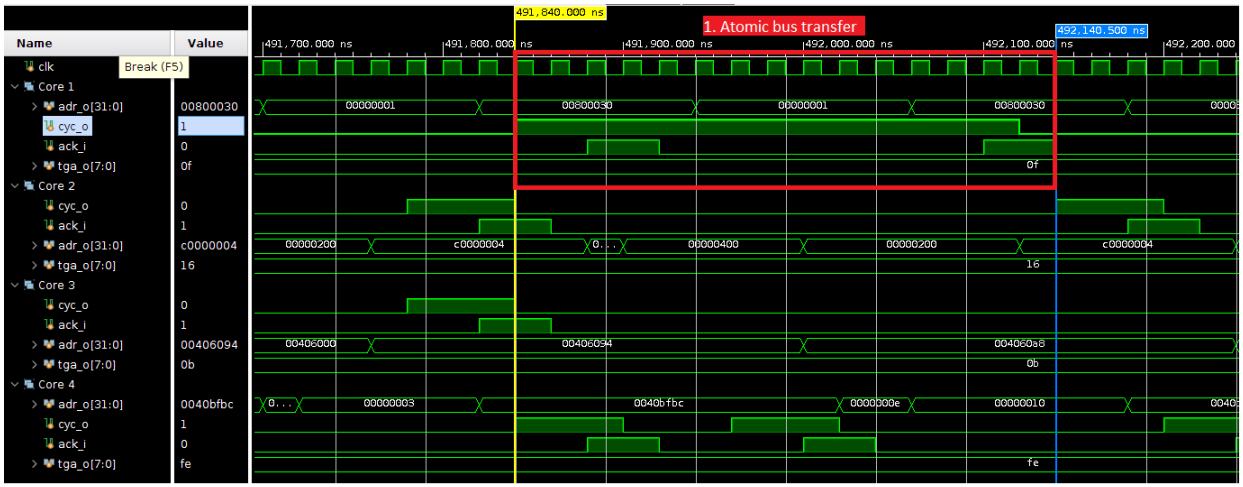


Figure 4.3: This shows an atomic bus mode transfer of core 1 (red rectangle).

The atomic transfer can be seen in detail in the signals of core 1 (Figure 4.3 Marker 1). It is an operation on a lock variable of the **prinftx** function. This shows the address of 0x800030. The WB master controller starts the atomic transfer mode by detecting a corresponding atomic instruction signal of a request. The transfer starts with a high on-the-cycle signal and

has the same procedure as a single non-pipelined signal until the acknowledged signal of the slave. The connections are not aborted with the receipt of the acknowledgement. It stays open until the next write bus operation. That can be seen in the address of the core 1 data bus signal. After the read of the lock variable, the address changes, and the core does other operations. The write cycle can be seen as the address changes back to 0x800030. After the second acknowledgement of the slave, which indicates the successful write operation, the atomic transfer is ended.

The edge cases for this atomic bus transfer implementation are:

- What happens if two or more masters simultaneously try an atomic bus transfer on the same address?
- What happens if no write operation follows after an atomic instruction signal?
- What happens if higher-priority bus requests try an atomic transfer to the same address as an ongoing lower-priority bus request?

To discuss the first edge case, two or more masters simultaneously try an atomic bus transfer on the same address. An atomic bus request is for the WB arbiter (see Section 3.1.1), no special case. The arbiter will grant access to the slave on the priority of the request. If both requests have the same priority, the request from the core with the lower core number wins. This core number is defined in hardware. If access is granted, the connection will be blocked until the next bus operation of the corresponding master. Therefore, the design makes only one atomic bus transfer to the same address possible.

What happens for the second edge case if the atomic bus transfer does not end with a write operation? The WB master controller (see Section 6.1.4) of the corresponding master will end the atomic bus transfer with the subsequent bus request of the master. This process ensures that the bus system is not locked forever because of a wrong usage in the code of the atomic instruction.

The third edge case is impossible per design, as an atomic bus transfer is not preemptive. Therefore, the higher-priority bus request must wait until the lower bus request is made. The WB arbiter (see Section 3.1.1) will not grant the connection, and therefore, this case is also avoided per design.

The correctness of the implementation of the atomic bus transfer is shown in the next section's measurements. The edge cases are covered by the design of the WB modules. The limitation of this implementation is that it works only for one read operation and one write operation on the bus. Therefore this atomic bus transfer can only be used for a lock variable other IPC concepts.

4.2.5 Interprocess mutual exclusive concept measurements

This section shows the newly implemented IPC features of the SmartOS multi-core version in hardware. The mutex and spinlock concept was measured via Cutecom and the oscilloscope in hardware. The message-passing concept was measured in hardware with the oscilloscope. The C-code for the measurements can be found in Section 6.1, Listing 6.1.

Mutex and spinlock demonstration with UART

The 4-core multi-core system with the SmartOS multi-core version was used for this measurement setup. To evaluate the functionality of the spinlock and mutex concept, the peripheral UART of the FPGA was used. To send data from a core to the UART, the WB crossbar opens a bus connection between the core and the UART (see Section 3.1.1). The UART hardware module has a receive and transfer buffer. To send data via the UART, the core writes data via the bus to the transfer buffer.

The writing in the transfer buffer in C-code is done with the **printfx** function. The **printfx** writes all characters of the string parameter of the function to the transfer buffer. As the UART transfer buffer has a fixed size, the **printfx** function puts characters in the transfer buffer until the transfer buffer is full. If the transfer buffer is full, the **printfx** waits until the UART hardware module sends the data from the transfer buffer and empties the buffer. Then, the **printfx** repeats the process until all string characters are written in the transfer buffer. The code for synchronised usage of **printfx** can be found in Listing 6.2. The code without IPC locking procedure is not included, as it is the same code only without using mutex and spinlock features.

In Figure 4.4, the reason why an IPC mechanism is essential for the usage of resources in a multi-core OS can be seen. Resources used without synchronised usage can corrupt the results (see Section 2.2.3). A multi-core system without a mutual exclusive usage of the UART peripheral can be seen on the left side of the figure. The **printfx** function uses the UART not mutually exclusive. Therefore, much of the information sent via the UART is corrupted.

This data corruption happens because every core can write data to the transfer buffer anytime if the WB bus connection to the UART is free. Therefore, two or more cores can simultaneously write data to the transfer buffer, corrupting the CuteCom Terminal's received data. A race condition (see Section 2.2.3) is caused by the missing synchronisation of the UART usage. The data can be in the wrong order or overwritten.

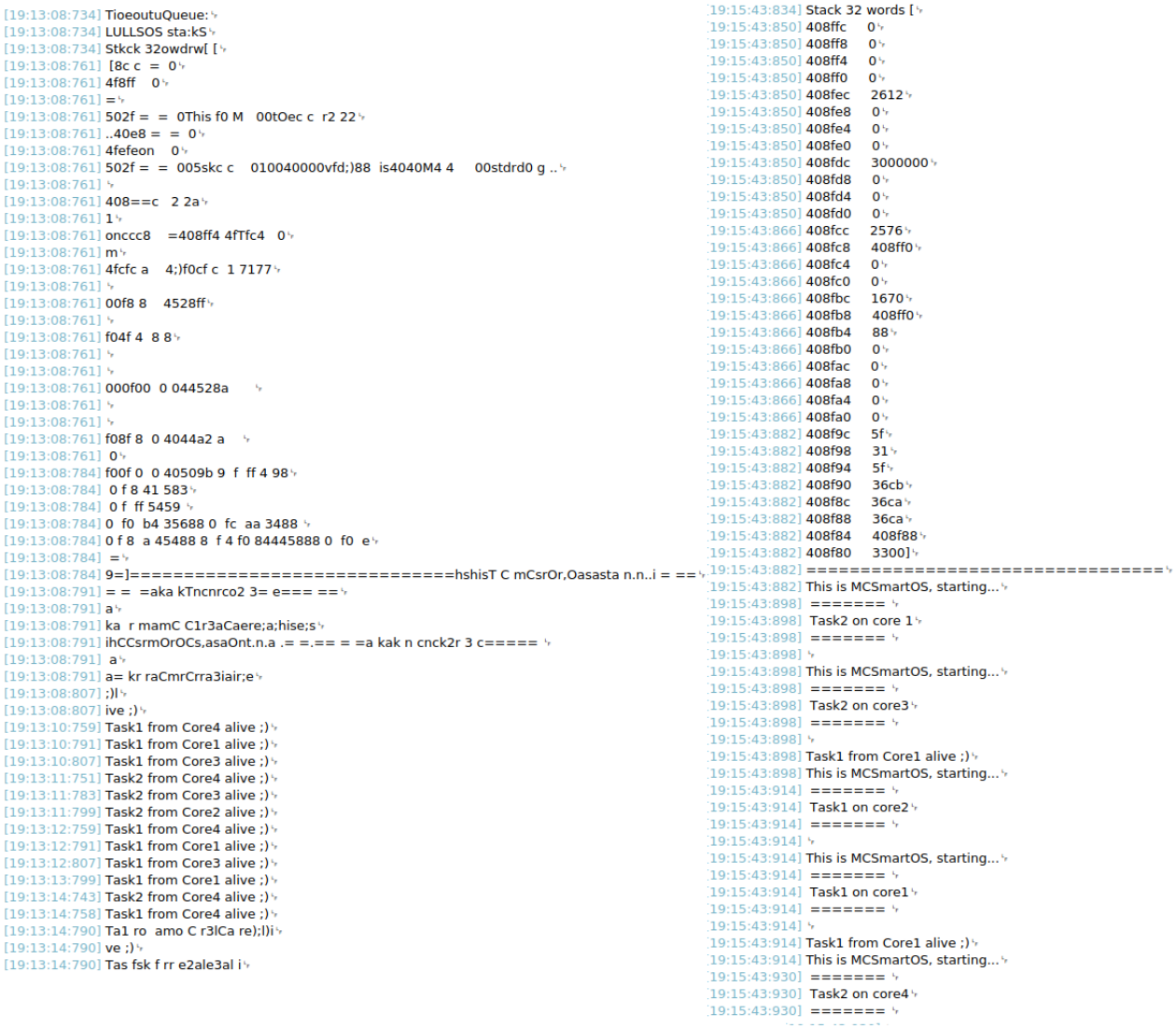


Figure 4.4: UART printfx output: On the left not synchronised versus right synchronised by mutex and spinlocks.

The correct data that should be sent can be seen on the right side in Figure 4.4. The upper message is the debug information for a core. It shows the ready queue, timeout queue, and task stacks. Other cores or tasks corrupt the information sent on the left side. The debug information of the OS is printed in the startup code, where all tasks, queues, and resources are initialised. Spinlocks are used in the synchronised startup code. The reason for this is that no tasks are active, and therefore, the mutex concept would not work. The following information that is printed is the startup information for the tasks. This startup information shows the core number of the running task and the corresponding task name.

The same uncorrupted information as on the left side can be seen on the right of Figure 4.4. Except for the critical factor, these **printfx** operations are adequately locked by the spinlock

(see Section 3.2.3) and mutex (see Section 3.2.4) concepts of the OS. For this example, two tasks per core were used. Every task triggers a corresponding LED and shows it is still alive via a **printfx** message. The mutex and spinlock operation uses the same lock variable for the **printfx** function.

In the locked **printfx** version, before the **printfx** is called, the task or startup code on a core tries to lock the lock variable. The lock variable is shared by all cores and located in the IPC memory (see Section 3.2.1). Ever access to this variable is done with the TSL instruction (see Section 2.2.3) to ensure atomic access in the core and the bus system. Therefore, no other task, regardless of which core it runs, can interfere with this operation. If the **printfx** is locked, only the owner of this lock can access the UART, and all data are sent uncorrupted to the CuteCom terminal. At the end of the **printfx** function, the lock variable is unlocked and used by others.

This section proves that the mutex and spinlock implementation works. The corresponding software and hardware concepts are implemented to ensure atomic access to the lock variable within the core and the bus system. The implemented atomic bus transfer shown in the simulation (see Section 4.2.4) works on the hardware.

Mutex demonstration via oscilloscope measurement

The measurement for this section uses the JA Pins of the FPGA to measure which task is the owner of a locked mutex. The code for this measurement can be seen in Listing 6.3. The mutex **mutex_leds** is used to secure mutual exclusive access to the two LEDS via the function **LEDsetState()**. Every task has a particular LED scheme. Therefore, LED1 is used as the first bit and LED2 as the second. LED1 and LED2 are directly connected to the JA Pin 1 and JA Pin 2 to measure the state of the LEDs with the oscilloscope.

Every core has one task that tries to set its unique binary-encoded LED state. The first cores task has the state LED1 off and LED2 off. The task LED state is equal to 00. The task of core 2 has the state 01, the core 3 task state is 10, and the core 4 task state is 11. To show the functionality of the mutex, every task tries to lock the **mutex_leds**.

If a task successfully locks the mutex, it sets LED1 and LED2 to its own defined state. After this, the task sleeps for a second to hold its state active. Afterwards, the task unlocks the **mutex_leds** and sleeps for another second. This sleeping increased the chance for other tasks to set their state, as every task has the same priority. As all task has the same priority, the task which first tries to lock the free mutex will win the race as shown in Section 3.2.4 in the code example Listing 3.8, a task yields after it fails to lock a mutex. This yield ensures that one task does not block the bus system, as the context switch takes time. This context switch allows other tasks to try to lock or unlock the mutex. If two tasks try to access the lock variable via the bus at the same time, the WB bus arbiter (see Section 6.1.4) decides who gets the access.

Figure 4.5 shows the measurement setup for the above-described mutex-LED state test. A two-channel Hantek Digital Storage oscilloscope is connected via its probes to jumper cables. These jumper cables are connected to the pins JA1 and JA2 of the FPGA Artix-7-Nexys-4. The oscilloscope settings are 800 milliseconds SEC/DIV and 2 Volts VOLTS/DIV.

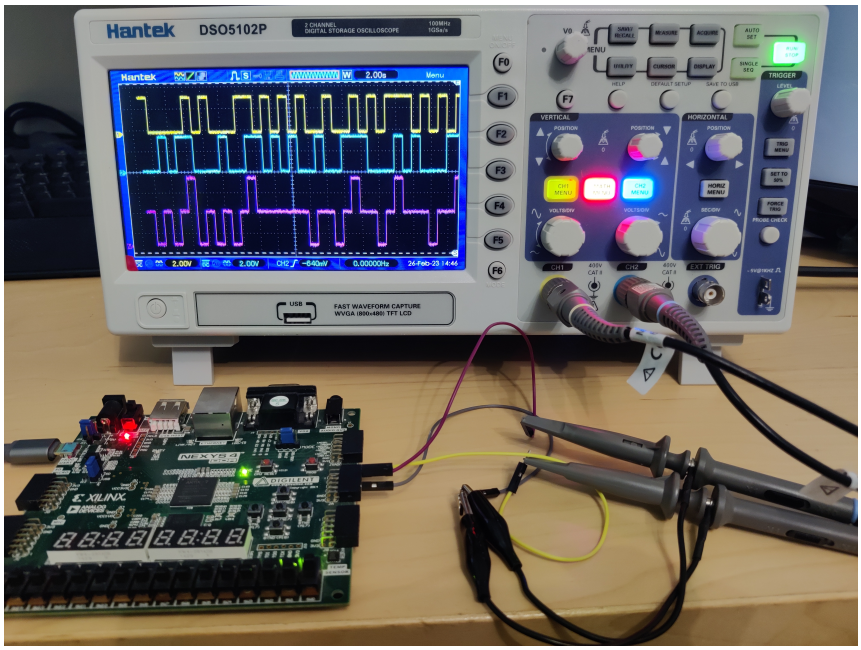


Figure 4.5: Measurement setup.

Figure 4.6 shows the measurement of the oscilloscope. The yellow signal is the state of LED1, the blue signal is the state of LED2, and the purple signal is the mathematical function of the signals LED1 + LED2. Above the signals, the current state is shown in white letters.

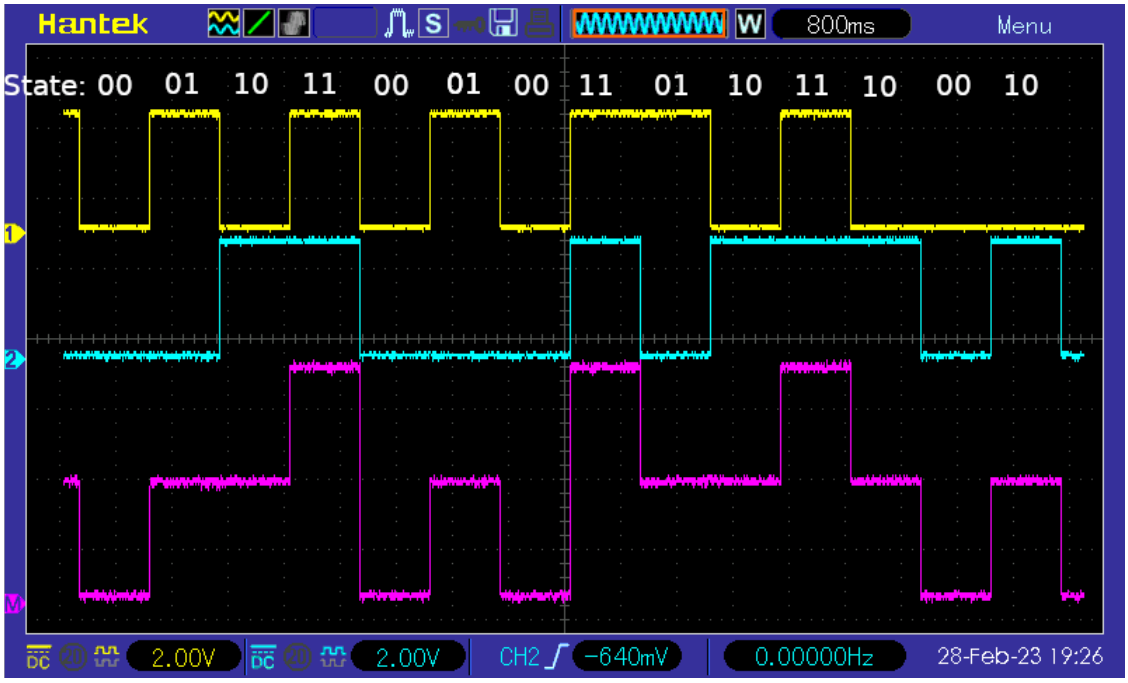


Figure 4.6: Oscilloscope measurement: mutex LED-states: Signals LED1 yellow, LED2 blue and math operation LED1 + LED2 purple.

In Figure 4.6, it can be seen that every task holds the state of the LED for a second and then lets other tasks change the state. As it can be seen, no task blocks the state, as all tasks have the same priority. The task of core 1 (State 00) got the mutex of the LEDs four times, the core 2 task (State 01) three times, the core three task (State 10) four times and the task of core 4 (State 11) three times.

This section shows that the new multi-core SmartOS's mutex feature works correctly, and the timings work as desired. The atomic bus transfer shown in Section 4.2.4 works without bugs on the actual hardware. All states of the LEDs in the measurement are precisely one second. That proves the TSL concept for the lock variables works, as no other task, regardless of which core it runs, can change the state of the mutex locked LEDs.

4.2.6 Message passing via oscilloscope measurement

The Section shows the implemented message-passing feature of the multi-core OS. The code for the measurement can be seen in Listing 6.1. In Figure 4.7, an overview of the tasks and functions can be seen, showing the observed measurement process in a simplified form. There are four tasks, all running on their core.

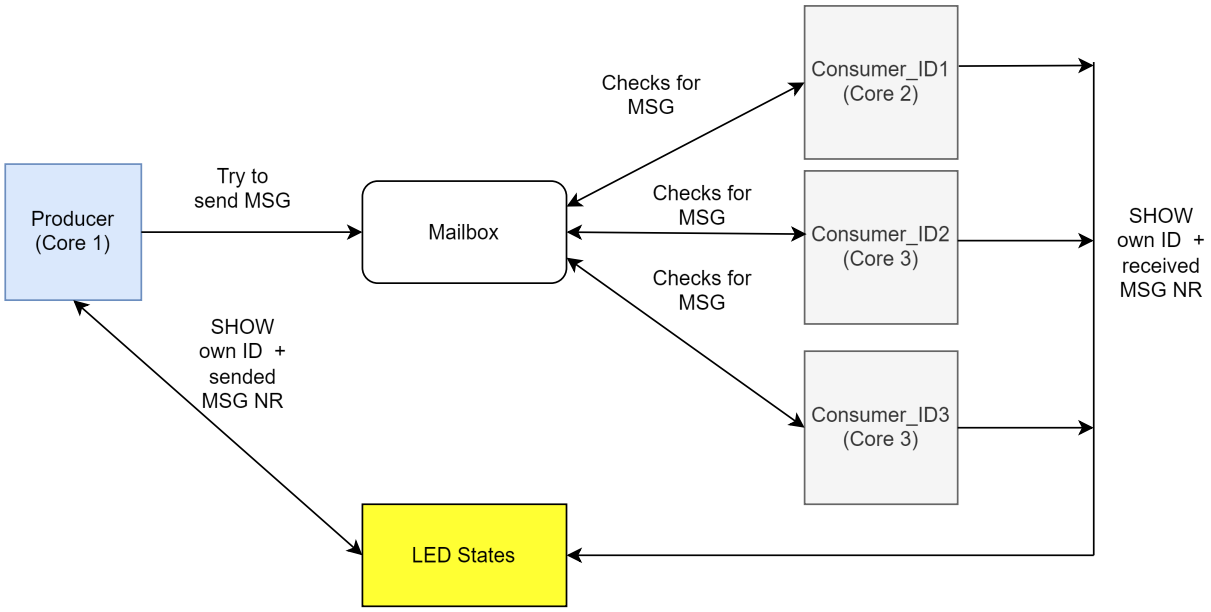


Figure 4.7: Overview of message passing measurement.

On core 1, a producer task sends a message to the consumer tasks on the other cores via a mailbox. Every message contains the destination ID and an ongoing number as a payload. This number, which ranges from 0 to 254, is used to identify messages.

The LEDs of the FPGA are used to measure this. LED1 shows the ID of the task in binary that receives or sends a message. LED2 shows the message number of the received or sent message in binary. Every LED is connected to a pin (JA) to measure the state of the LED with the oscilloscope. If the producer sends a message successfully, its ID and message number are shown as 8-bit binary data on the LEDs. If a consumer task receives a message, it shows its ID in binary on LED1 and the message number received on LED2. The length of one bit is 25 milliseconds. Therefore, all 8 Bits have a length of 200 ms. This time was chosen to make it easy to see the multiple states on the oscilloscope.

All tasks have the same priority: to test the implemented message-passing concept. The mailbox can hold five messages, and the producer only sends a message if a place is empty. Therefore, it retries to send a message until a space is free. The producer sends every second a new message if a space is free. All consumers also check the mailbox in an interval of one second. This interval was chosen to better visualise the results on the oscilloscope.

In Figure 4.8, the start- and end-sequences of data output can be observed. To better identify the shown LED states of the task. Every message starts and ends with toggling LED1 and LED2 three times. The toggling interval is 12.5 milliseconds and, therefore, shorter than a bit interval. This is done to better identify start- and end-sequences on the oscilloscope. In this start sequence, LED1 and LED2 always have an opposite state. This means if LED1 is on, LED2 is off. Between the start and end sequence on LED1, the ID of the corresponding task is shown as 8-bit binary data and the message number on LED2.

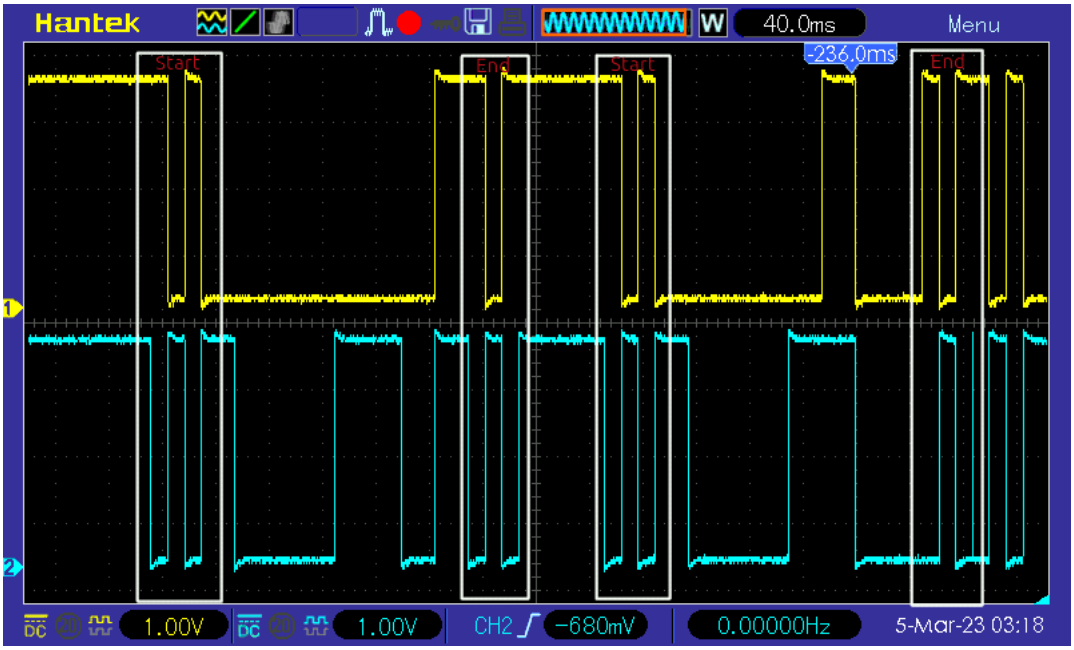


Figure 4.8: Oscilloscope measurement: Message passing LED-states: Shows Start and End LED-states in white rectangles.

Figure 4.8 shows two messages in the white rectangles. The end and start sequence can be observed. This implementation makes it possible to distinguish between different signals.

Three message signals can be seen in Figure 4.9. The first message is from the producer with ID 1. The producer sends the message with the number 1. The second message signal is from a consumer with ID 3. Consumer 3 receives the message from the producer. The third signal shown is from the consumer with ID 2, who received the message with the number zero.

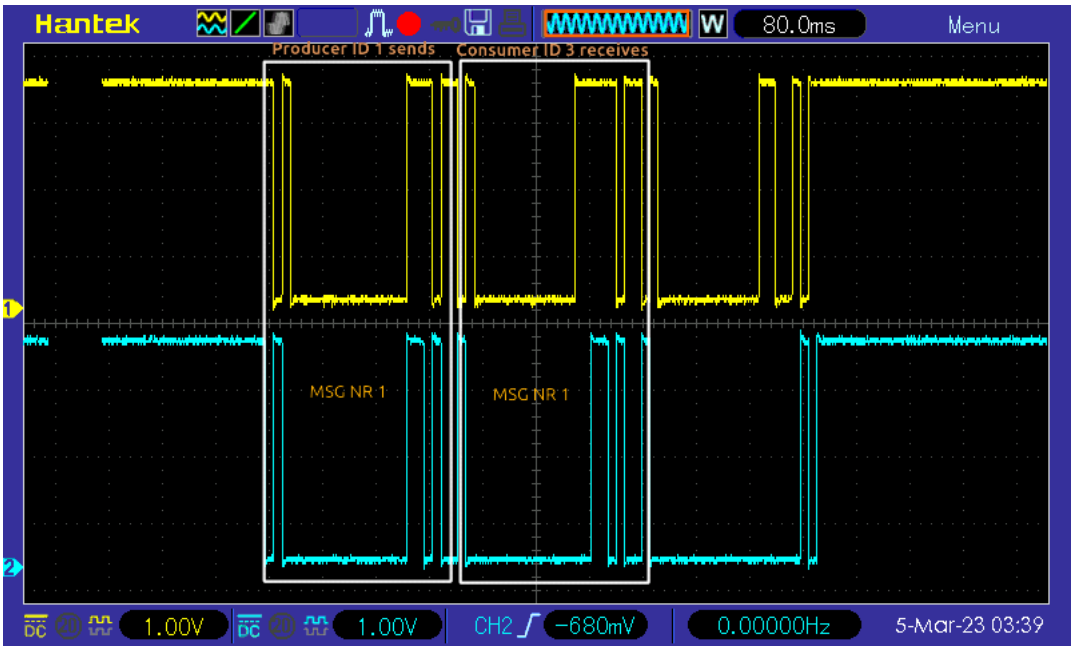


Figure 4.9: Oscilloscope measurement: Message passing LED-states :
Signals LED1 yellow shows ID, LED2 blue shows message number .
Three messages can be seen, but only two are described.

These measurements show that the message-passing concepts work well across all cores. All messages sent from a consumer to a producer were received without corruption. In Section 3.2.5, the detailed implementation of the message-passing concept can be seen. The message-passing object can only accessed via the send-and-receive function. Therefore, a message's data can only be accessed with a locked mutex. This implementation allows data to be shared across tasks and cores without race conditions.

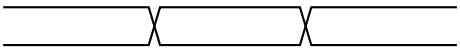
4.2.7 Measurement and simulation discussion

This chapter evaluated the implemented multi-core system and its modules for scalability. The functionality of the implemented WB crossbar with the integrated arbiter was shown in the simulation. The proof of concept of the multi-core system and the corresponding SmartOS multi-core system was provided in the measurements section of this chapter. The measurements focused on the newly implemented IPC features of the SmartOS. This was done because a main part of this master thesis was to generate a good synchronised multi-core OS.

CHAPTER 5

Conclusion and Future Work

This chapter concludes the master's thesis by summarising the results from the previous chapters. Future work will list all features that could not be implemented and would improve the multi-core OS and multi-core system.



5.1 Conclusion

This master thesis evolved the SmartOS to a multi-core OS and implemented the corresponding multi-core system. A literature research was conducted to dive deeply into bus systems, bus data transfer modes and bus topologies. Different kinds of multi-core OS concepts were explored. IPC concepts for embedded systems were elevated by their pros and cons to handle synchronisation challenges. The related multi-core systems were evaluated to gain knowledge for the master thesis's implemented multi-core system and multi-core OS.

The heart of a multi-core system is the way to exchange information. The chosen architecture for this is the WB bus. In detail, a WB crossbar with integrated arbiter. The WB bus supports user-defined arbiter schemes and user-defined tag signals. This was used to let the arbiter decide on the task priority of the cores, which would get the connection. For this WB bus, a RMW bus mode was implemented to make atomic bus transfers. This atomic bus transfer is used for the implemented IPC concepts.

The crossbar was implemented as a topology for a flexible number of cores. The trade-off between bandwidth and used area works fine. Section 4.1.1 evaluates the used components of the FPGA for a four-core and a six-core system. As seen, the FPGA would support more cores than the implemented four-core system, and the resource consumption of the crossbar is acceptable.

A static priority arbiter is used for the arbiter. The crossbar was implemented to be as flexible as possible to exchange the arbiter. Therefore, different bus arbiters can be used in the future.

The arbiter uses the priority of the current task on the core as a criterion for the next connection. The dynamic interfaces for the wishbone make it easy to increase or decrease the number of cores. The whole implementation is designed to be as flexible as possible.

The used OS concept is that every core has its own OS, and all cores use the same OS code. However, every core has its own OS data structures. The message-passing concept was implemented to exchange data between tasks. The mutex and spinlock concept was used to secure critical regions. Section 4.2.5 shows that the implemented IPC concepts work well, and the synchronisation between the cores is a complete success.

The goals for master thesis (see Section 1.2) were:

- Evaluate other multi-core systems, bus systems, OS concepts and IPC.
- Implement a hardware multi-core environment with these criteria:
  - This hardware implementation has to be easily expandable. Therefore, the number of cores should be easily changed.
  - System Verilog has to be the used Hardware Description language, as modules already exist in this language.
  - The bus arbiter should be real-time capable, as in further usage, real-time is necessary.
  - The implemented modules should be easy to use.
- Implement a multi-core SmartOS version.

The first goal was achieved in Chapter 2, where a deep literature review was done for the hardware/software co-design. For the second goal, an expandable, multi-core system was implemented in System Verilog (see Section 3.1), and with a real-time capable arbiter, this goal is fulfilled. A multi-core version of SmartOS (see Section 3.2) was implemented, and therefore all goals were achieved.

5.2 Future work

In this master's thesis, different ideas arose that could not be included because they would have been beyond the scope of this master's thesis. Possible two improvements would be:

WB bus systems, only one arbiter was made. Therefore, the system could be extended with the explained bus arbiter of the bus arbiter section. Therefore, different metric measurements and analyses could be made. The WB crossbar only supports two bus modes. The bus transfer modes could be extended to support all existing modes of the WB bus.

The SmartOS now supports **Resource and Events only** for a single core. These features could be expanded to achieve a multi-core functionality for the event and resource system. The implemented SmartOS version only supports four cores. The code of the OS could be improved with macros so that the code for more cores is created only by changing a number.

Bibliography

[1] L. Harvie, "Breaking barriers with multicore processors in embedded systems," *Medium*, 2023. [Online]. Available: https://medium.com/@lanceharvieruntime/breaking-barriers-with-multicore-processors-in-embedded-systems-753a790fe240 → [p1]

[2] T. Scheipel, F. Angermair, and M. Baunach, "moreMCU: A Runtime-reconfigurable RISC-V Platform for Sustainable Embedded Systems," in *Proceedings of the 25th Euromicro Conference on Digital System Design (DSD) - under review*, Euromicro. Maspalomas, Spain: IEEE, Sep. 2022. → [p1]

[3] *Nexys 4 Artix-7 FPGA Trainer Board*. [Online]. Available: https://digilent.com/reference/programmable-logic/nexys-4/start → [p2], [p55]

[4] T. Scheipel, L. Batista Ribeiro, T. Sagaster, and M. Baunach, "SmartOS: An OS Architecture for Sustainable Embedded Systems," in *Tagungsband des FG-BS Frühjahrstreffens*. Hamburg, Germany: Gesellschaft für Informatik e.V., Mar. 2022, pp. 1–10. → [p5], [p19], [p20], [p93]

[5] S. Pasricha and N. Dutt, *On-Chip Communication Architectures*. Morgan Kaufmann Publishers, 2008. → [p6], [p7], [p8], [p12], [p13], [p14], [p15], [p16], [p93]

[6] A. S. Tannenbaum, *Structured Computer Organization*, 5th ed. Pearson Education, Inc, 2006. → [p9], [p93]

[7] P. Bajaj and D. Padole, "Arbitration schemes for multiprocessor shared bus," *InTech*, 2011. [Online]. Available: http://www.intechopen.com/books/new-trends-and-developments-in-automotive-system-engineering/arbitration-schemes-for-multiprocessor-shared-bus → [p10], [p59]

[8] F. Hebbache, "Work-conserving dynamic TDM-based memory arbitration for multi-criticality real-time systems," Ph.D. dissertation, Université Paris Saclay (COmUE), 2020. → [p10], [p11]

[9] K. Lahiri, A. Raghunathan, and G. Lakshiminarayan, "Lotterybus: A new high-performance communicationarchitecture for system-on-chip designs," *Design Automation Conference,*, 2001. → [p11]

[10] D. Conrads, *Datenkommunikation: Verfahren - Netze - Dienste.* Friedrich Vieweg und Sohn Verlag, 1993. → [p12]

[11] W. D. Peterson, "Wishbone b4, wishbone system-on-chip(soc)interconnection architecture for portable ip cores," 2010. [Online]. Available: http://cdn.opencores.org/downloads/wbspec_b4.pdf → [p15], [p17]

[12] *Nios® V Processor Software Developer Handbook*. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/743810/24-1/overview-of-embedded-processor-development.html → [p16]

[13] *Avalon Interface Specifications* , 2022nd ed. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf → [p16]

[14] *AMBA® 5 CHI Architecture Specification*. [Online]. Available: https://developer.arm.com/documentation/ihi0050/latest → [p16]

[15] *AMBA AHB Protocol Specification*. [Online]. Available: https://developer.arm.com/documentation/ihi0033/latest/ → [p17]

[16] *AMBA APB Protocol Specification*. [Online]. Available: https://developer.arm.com/documentation/ihi0024/latest/ → [p17]

[17] *AMBA AXI and ACE Protocol Specification*. [Online]. Available: https://developer.arm.com/documentation/ihi0051/latest/ → [p17]

[18] A. S. Tannenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Pearson Education Limited, 2015. → [p20], [p21], [p22], [p24], [p25], [p26], [p30], [p31], [p33], [p34], [p37], [p93], [p97]

[19] D. A. Patterson and A. Waterman, *The RISC-V reader: an open architecture atlas*, first edition 1.0.0 ed. Strawberry Canyon LLC, 2017. → [p27], [p93]

[20] M. Baunach, "Real time operating systems vo," 2020. → [p28], [p29], [p30]

[21] A. Wieder and B. B. Brandenburg, "On spin locks in autosar: Blocking analysis of fifo, unordered, and priority-ordered spin locks," 2013. [Online]. Available: https://www.mpi-sws.org/tr/2013-005.pdf → [p32], [p33], [p97]

[22] W. Stallings, *Operating Systems Internals and Design Principles*, 9th ed. Pearson Education Limited, 2018. → [p34], [p35], [p36], [p38], [p40], [p41], [p43], [p93], [p95], [p97]

[23] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Moran Kaufmann, 2019. → [p46], [p47], [p48], [p93]

[24] F. Conti, R. Schilling, P. D. Schiavone, A. Pullini, D. Rossi, F. K. Gürkaynak, M. Muehlberghuber, M. Gautschi, I. Loi, G. Haugou, S. Mangard, and L. Benini, "An iot endpoint system-on-chip for secure and energy-efficient near-sensor analytics," 2017. → [p49], [p93]

[25] A. Pullini, F. Conti, D. Rossi, I. Loi, M. Gautschi, and L. Benini, "A heterogeneous multicore system on chip for energy efficient brain inspired computing," 2018. → [p50], [p93]

[26] A. Pullini, D. Rossi, I. Loi, A. D. Mauro, and L. Benini, "Mr. wolf: A 1 gflop/s energy-proportional parallel ultra low power soc for iot edge processing," 2018. → [p51], [p93]

[27] F. Zarubaand, F. Schuikiand, T. Hoefler, and L. Benini, "Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads," 2020. → [p52], [p93]

[28] T. Scheipel, "Advances in dynamic and reconfigurable embedded systems design," Ph.D. dissertation, Graz University of Technology, 2022. → [p61]

[29] *Hantek DSO5102P two-channel digital oscilloscope*, 2020th ed. [Online]. Available: https://www.hantek.com/products/detail/96 → [p75]

[30] *Linux Terminal CuteCom*, 2009th ed. [Online]. Available: https://cutecom.sourceforge.net/ → [p75]

List of Figures

List of Tables

List of Listings

List of Abbreviations

AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
AMO	Atomic Memory Operations
APB	Advanced Peripheral Bus
ASB	Advanced System Bus
AUTOSAR	Automotive Open System Architecture
AXI	Advanced eXtensible Interface
BRAM	Block RAM
ChB	Chained Blocking
CHI	Coherent Hub Interface
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CSR	Control and Status Register
CSRs	Control and Status Registers
DB	Direct blocking
DMA	Direct Memory Access
DSMP	Distributed Shared Memory Multiprocessor
DSP	Digital Signal Processing
DSP	Digital Signal Processor
FF	Flip Flop
FIFO	First In First Out
FLL	frequency-locked loops
FP	Floating Point
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
GPIO	General-Purpose Input/Output
HLP	Highest Locker Protocol
IO	Input Output
IoT	Internet of Things
IPC	Interprocess Communication

ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group
KiB	Kibibyte
LUT	Look-up Table
MAC	Multiplier–accumulator
MCU	Microcontroller Unit
MM	memory mapped
MMCM	Mixed-Mode Clock Manager
mutex	Mutual Exclusion
MUXES	Multiplexer
NUMA	Nonuniform Memory Access
OS	Operating System
PCP	Priority Ceiling Protocol
PIP	Priority Inheritance Protocol
PTB	Push-trough-blocking
RAM	Random Access Memory
RMW	Read-Modify-Write
ROM	Read Only Memory
RR	Round Robin
RTOS	Real Time Operating System
SMP	Symmetric Multiprocessors
SoC	System on Chip
SOPC	System-On-a-Programmable-Chip
SP	scratch-pads
ST	streaming
TCDM	Tightly Coupled Data Memory
TDM	Time Division Multiplexed
TSL	Test and Set Lock
UART	Universal Asynchronous Receiver-Transmitter
UMA	Uniform Memory access
WB	Wishbone
XCHG	eXCHanGe

CHAPTER 6

Appendix

6.1 Code Appendix

6.1.1 Message passing example code

```
1  #include <kernel/smartos.h>
2  #include <drv/led.h>
3  #include <arch/message_passing.h>
4  #include <libs/printfx.h>
5  #include <string.h>
6  #include <arch/mutex.h>
7
8  OS_CREATE_MESSAGE_MAILBOX_STRUCT(box1);
9  OS_CREATE_MUTEX(leds);
10
11 OS_TASK_CONSTR_CORE1(core1_producer_id1, 600, 15);
12 OS_TASK_CONSTR_CORE2(core2_consumer_id2, 600, 15);
13 OS_TASK_CONSTR_CORE3(core3_consumer_id3, 600, 15);
14 OS_TASK_CONSTR_CORE4(core4_consumer_id4, 600, 15);
15
16 void __attribute__((optimize("-O0"))) showOnLEDsAsBit(uint8_t id,
       uint8_t msg_nr)
17 {
18
19    mutex_lock(&mutex_leds);
20    uint8_t stb_eb_cyc = 3;
21    uint8_t i = 0;
22    Time_t later;
23    Time_t interval = (Time_t)(getOneSecond())/40;
24
25    while(i < stb_eb_cyc)
26    {
27       getCurrentTime(&later);
```

```
28
29          if(i % 2)
30          {
31             LEDsetState(LED_01,LED_OFF);
32             LEDsetState(LED_02,LED_ON);
33          }
34          else
35          {
36             LEDsetState(LED_01,LED_ON);
37             LEDsetState(LED_02,LED_OFF);
38          }
39 #ifdef SIM
40          yield();
41 #else
42          later += interval/2;
43          sleepUntil(later);
44 #endif
45          i++;
46       }
47       i = 0;
48       while( i < 8)
49       {
50
51          uint8_t bit_id = (id >> (7 - i)) & 0x1;
52          LEDsetState(LED_01,bit_id);
53          uint8_t bit_msgnr = (msg_nr >> (7 - i)) & 0x1;
54          LEDsetState(LED_02,bit_msgnr);
55
56          getCurrentTime(&later);
57          #ifdef SIM
58          yield();
59          #else
60          later += interval;
61          sleepUntil(later);
62          #endif
63          i++;
64       }
65
66       i = 0;
67       while(i < stb_eb_cyc)
68       {
69          getCurrentTime(&later);
70
71          if(i % 2)
72          {
73             LEDsetState(LED_01,LED_OFF);
```

```
74            LEDsetState(LED_02,LED_ON);
75         }
76         else
77         {
78            LEDsetState(LED_01,LED_ON);
79            LEDsetState(LED_02,LED_OFF);
80         }
81 #ifdef SIM
82         yield();
83 #else
84         later += interval/2;
85         sleepUntil(later);
86 #endif
87
88         i++;
89      }
90      LEDsetState(LED_01,LED_ON);
91      LEDsetState(LED_02,LED_ON);
92      mutex_unlock(&mutex_leds);
93 }
94 OS_TASKENTRY(core1_producer_id1)
95 {
96      Time_t later;
97      Time_t interval = (Time_t)(getOneSecond());
98
99      uint8_t source_id = 1;
100     uint8_t dest_id = 2;
101     uint8_t msg_nr = 0;
102     char send_data[8] = "id2_000\0";
103
104     mutex_lock_printfx();
105 #ifdef SIM
106     uint8_t i = 0;
107     while(i <40)
108     {
109        i++;
110        LEDtoggle(LED_16);
111     }
112 #else
113     printfx("This is MCSmartOS, starting...\n ======= \n Producer id1
            on core 1\n ======= \n\n");
114
115 #endif
116     mutex_unlock_printfx();
117     while (1)
118     {
```

```
119         //print what is send and to who
120         mutex_lock_printfx();
121         #ifndef SIM
122
123            printfx("Producer id1 from core1 try to send data to dest id
                   %d : %s ;)\n",dest_id,send_data );
124         #else
125            i = 0;
126            while(i <40)
127            {
128               i++;
129            }
130          #endif
131         mutex_unlock_printfx();
132         //sending the message
133         while(send(&message_mailbox_struct_box1,dest_id,source_id,
               send_data))
134         {
135             mutex_lock_printfx();
136
137             #ifndef SIM
138                  printfx("Producer id1 from core1 mailbox full retry\n"
                         );
139             #else
140                  i = 0;
141                  while(i <40)
142                  {
143                     i++;
144                     LEDtoggle(LED_16);
145                  }
146             #endif
147             mutex_unlock_printfx();
148
149             getCurrentTime(&later);
150             #ifndef SIM
151               later += interval/10; //wait 100ms
152               #else
153               later += 10000;
154             #endif
155
156             sleepUntil(later);
157         }
158         showOnLEDsAsBit(source_id,msg_nr);
159
160         msg_nr++;
161         if(msg_nr == 254)
```

```
162             msg_nr = 0;
163         mutex_lock_printfx();
164         #ifndef SIM
165         printfx("Producer id1 from core1 send successfully\n");
166         #else
167         i = 0;
168         while(i <40)
169         {
170             i++;
171             LEDtoggle(LED_16);
172         }
173         #endif
174         mutex_unlock_printfx();
175         //change paylod of message for different tasks
176         if(dest_id == 2)
177         {
178             dest_id = 3;
179 
180         }
181         else if(dest_id == 3)
182         {
183             dest_id = 4;
184         }
185         else if(dest_id == 4)
186         {
187             dest_id = 2;
188         }
189         send_data[2] = dest_id + 0x30 ;
190         uint8_t digit_100 =  msg_nr/100 ;
191         uint8_t digit_10 =  msg_nr/10 - digit_100*10 ;
192         uint8_t digit_1 =  msg_nr - digit_10*10 - digit_100 *100 ;
193         send_data[4] = digit_100 + 0x30;
194         send_data[5] = digit_10 + 0x30;
195         send_data[6] = digit_1 + 0x30;
196         getCurrentTime(&later);
197         #ifndef SIM
198         later += interval;
199         #else
200         later += 10000;
201         #endif
202         sleepUntil(later);
203     }
204 }
205 /* Consumer with id 2 on Core 2 */
206 OS_TASKENTRY(core2_consumer_id2)
207 {
```

```
208    Time_t later;
209    Time_t interval = (Time_t)(getOneSecond() * 1);
210    uint8_t source_id = 0xFF;
211    uint8_t own_id = 2;
212    char receive_data[8]  = "FFFFFFFF";
213
214    mutex_lock_printfx();
215    #ifndef SIM
216     printfx("This is MCSmartOS, starting...\n ======= \n Consumer id2
           on core2\n ======= \n\n");
217    #else
218       uint8_t i = 0;
219       while(i <40)
220       {
221          i++;
222          LEDtoggle(LED_15);
223       }
224    #endif
225    mutex_unlock_printfx();
226
227    while (1)
228    {
229      //print if message was received or show no data was send
230      if(receive(&message_mailbox_struct_box1,own_id,&source_id,
           receive_data))
231      {
232          mutex_lock_printfx();
233          #ifndef SIM
234             printfx("Consumer id2 from core 2 got data from %d : %s \
                  n",source_id,receive_data);
235          #else
236             i = 0;
237             while(i <40)
238             {
239                i++;
240                LEDtoggle(LED_15);
241             }
242          #endif
243          mutex_unlock_printfx();
244
245          uint8_t msg_nr = (receive_data[4] - 0x30) * 100;
246          msg_nr += (receive_data[5] - 0x30) * 10;
247          msg_nr += (receive_data[6] - 0x30);
248
249           mutex_lock_printfx();
250           printfx("msgnr is %d , id is %d \n",msg_nr,own_id);
```

```
251             mutex_unlock_printfx();
252            showOnLEDsAsBit(own_id,msg_nr);
253
254          }
255
256         else
257         {
258             mutex_lock_printfx();
259             #ifndef SIM
260                printfx("Consumer id2 from core 2 got no data ;(\n");
261             #else
262                i = 0;
263                while(i <40)
264                {
265                   i++;
266                    LEDtoggle(LED_15);
267                }
268             #endif
269             mutex_unlock_printfx();
270         }
271         getCurrentTime(&later);
272          #ifndef SIM
273          later += interval;
274          #else
275          later += 10000;
276          #endif
277         sleepUntil(later);
278        }
279 }
280
281 /* Consumer with id 3 on Core 3 */
282 OS_TASKENTRY(core3_consumer_id3)
283 {
284    Time_t later;
285    Time_t interval = (Time_t)(getOneSecond() * 1);
286    uint8_t source_id = 0xFF;
287    uint8_t own_id = 3;
288    char receive_data[8] = "FFFFFFFF";
289
290    mutex_lock_printfx();
291    #ifndef SIM
292       printfx("This is MCSmartOS, starting...\n ======= \n Consumer
           id3 on core3\n ======= \n\n");
293    #else
294       uint8_t i = 0;
295       while(i <40)
```

```
296         {
297             i++;
298             LEDtoggle(LED_14);
299         }
300     #endif
301     mutex_unlock_printfx();
302     while (1)
303     {
304         //get message if is available
305         if(receive(&message_mailbox_struct_box1,own_id,&source_id,
                receive_data))
306         {
307             mutex_lock_printfx();
308             #ifndef SIM
309                 printfx("Consumer id3 from core3 go data from %d : %s \n"
                        ,source_id,receive_data);
310             #else
311                 i = 0;
312                 while(i <40)
313                 {
314                     i++;
315                      LEDtoggle(LED_14);
316                 }
317             #endif
318             mutex_unlock_printfx();
319 
320             uint8_t msg_nr = (receive_data[4] - 0x30) * 100;
321             msg_nr += (receive_data[5] - 0x30) * 10;
322             msg_nr += (receive_data[6] - 0x30);
323              mutex_lock_printfx();
324              printfx("msgnr is %d , id is %d \n",msg_nr,own_id);
325              mutex_unlock_printfx();
326             showOnLEDsAsBit(own_id,msg_nr);
327 
328         }
329         else
330         {
331           mutex_lock_printfx();
332           #ifndef SIM
333               printfx("Consumer id3 from core 3 got no data ;(\n");
334           #else
335               i = 0;
336               while(i <40)
337               {
338                   i++;
339                   LEDtoggle(LED_14);
```

```
340             }
341           #endif
342           mutex_unlock_printfx();
343         }
344
345         getCurrentTime(&later);
346         #ifndef SIM
347         later += interval;
348         #else
349         later += 10000;
350         #endif
351         sleepUntil(later);
352       }
353 }
354 /* Consumer with id 4 on Core 4 */
355 OS_TASKENTRY(core4_consumer_id4)
356 {
357    uint8_t source_id = 0xFF;
358    uint8_t own_id = 4;
359    char receive_data[8] = "FFFFFFFF";
360    Time_t later;
361    Time_t interval = (Time_t)(getOneSecond() * 1);
362
363    mutex_lock_printfx();
364    #ifndef SIM
365       printfx("This is MCSmartOS, starting...\n ======= \n Consumer
           id4 on core4\n ======= \n\n");
366    #else
367       uint8_t i = 0;
368       while(i <40)
369       {
370          LEDtoggle(LED_13);
371          i++;
372       }
373    #endif
374    mutex_unlock_printfx();
375    while (1)
376    {
377       if(receive(&message_mailbox_struct_box1,own_id,&source_id,
           receive_data))
378       {
379          mutex_lock_printfx();
380          #ifndef SIM
381             printfx("Consumer id4 from core3 go data from %d : %s \n"
                  ,source_id,receive_data);
382          #else
```

```
383             i = 0;
384             while(i <40)
385             {
386                i++;
387                LEDtoggle(LED_13);
388             }
389          #endif
390          mutex_unlock_printfx();
391
392          uint8_t msg_nr = (receive_data[4] - 0x30) * 100;
393          msg_nr += (receive_data[5] - 0x30) * 10;
394          msg_nr += (receive_data[6] - 0x30);
395          mutex_lock_printfx();
396          printfx("msgnr is %d , id is %d \n",msg_nr,own_id);
397          mutex_unlock_printfx();
398          showOnLEDsAsBit(own_id,msg_nr);
399       }
400       else
401       {
402         mutex_lock_printfx();
403         #ifndef SIM
404         printfx("Consumer id4 from core 3 got no data ;(\n");
405         #else
406             i = 0;
407             while(i <40)
408             {
409                i++;
410                LEDtoggle(LED_13);
411             }
412          #endif
413         mutex_unlock_printfx();
414       }
415       getCurrentTime(&later);
416       #ifndef SIM
417       later += interval;
418       #else
419       later += 10000;
420       #endif
421       sleepUntil(later);
422    }
423 }
```

Listing 6.1: Testcode for message passing with one producer and three consumer.

6.1.2 Mutex printfx example code

```
#include <kernel/smartos.h>
#include <drv/led.h>
#include <arch/mutex.h>
#include <libs/printfx.h>
//Tasks creation for the differnt cores
OS_TASK_CONSTR_CORE1(task1_core1, 600, 15);
OS_TASK_CONSTR_CORE1(task2_core1, 600, 15);

OS_TASK_CONSTR_CORE2(task1_core2, 600, 22);
OS_TASK_CONSTR_CORE2(task2_core2, 600, 18);

OS_TASK_CONSTR_CORE3(task1_core3, 600, 11);
OS_TASK_CONSTR_CORE3(task2_core3, 600, 14);

OS_TASK_CONSTR_CORE4(task1_core4, 600, 12);
OS_TASK_CONSTR_CORE4(task2_core4, 600, 15);

OS_TASKENTRY(task1_core1)
{
//for simulation skip this as printfx takes too long and other
    sleeping invertval
#ifndef SIM
   mutex_lock_printfx();
   printfx("This is MCSmartOS, starting...\n ======= \n Task1 on
       core1\n ======= \n\n");
   mutex_unlock_printfx();
#endif
   Time_t later;
   Time_t interval = (Time_t)(getOneSecond() * 5);

   while (1)
   {

      getCurrentTime(&later);
      mutex_lock(&mutex_LED1);
      LEDtoggle(LED_01);
      mutex_unlock(&mutex_LED1);
#ifndef SIM
      mutex_lock_printfx();
      printfx("Task1 from Core1 alive ;)\n" );
      mutex_unlock_printfx();
      later += interval;
#else
      later += 80000; // 80 000 ns = 80 us
```

```
43 #endif
44       sleepUntil(later);
45     }
46 }
47
48 OS_TASKENTRY(task2_core1)
49 {
50 //for simulation skip this as printfx takes too long
51 #ifndef SIM
52    mutex_lock_printfx();
53    printfx("This is MCSmartOS, starting...\n ======= \n Task2 on core
          1\n ======= \n\n");
54    mutex_unlock_printfx();
55 #endif
56    Time_t later;
57    Time_t interval = (Time_t)(getOneSecond() * 6);
58
59    while (1)
60    {
61       getCurrentTime(&later);
62       LEDtoggle(LED_02);
63 #ifndef SIM
64       mutex_lock_printfx();
65       printfx("Task1 from Core1 alive ;)\n" );
66       mutex_unlock_printfx();
67
68       later += interval;
69 #else
70       later += 70000;
71 #endif
72       sleepUntil(later);
73    }
74 }
75 OS_TASKENTRY(task1_core2)
76 {
77 #ifndef SIM
78    mutex_lock_printfx();
79    printfx("This is MCSmartOS, starting...\n ======= \n Task1 on
          core2\n ======= \n\n");
80    mutex_unlock_printfx();
81 #endif
82    Time_t later;
83    Time_t interval = (Time_t)(getOneSecond() * 2);
84    while (1)
85    {
86      getCurrentTime(&later);
```

```
87      LEDtoggle(LED_03);
88 #ifndef SIM
89      mutex_lock_printfx();
90      printfx("Task1 from Core1 alive ;)\n" );
91      mutex_unlock_printfx();
92       later += interval;
93 #else
94      later += 80000; // 60 000 ns = 60 us
95 #endif
96      sleepUntil(later);
97     }
98 }
99 OS_TASKENTRY(task2_core2)
100 {
101 #ifndef SIM
102    mutex_lock_printfx();
103    printfx("This is MCSmartOS, starting...\n ======= \n Task2 on
          core2\n ======= \n\n");
104    mutex_unlock_printfx();
105 #endif
106    Time_t later;
107    Time_t interval = (Time_t)(getOneSecond() * 3);
108
109    while (1)
110    {
111       getCurrentTime(&later);
112       LEDtoggle(LED_04);
113
114 #ifndef SIM
115       mutex_lock_printfx();
116       printfx("Task2 from Core2 alive ;)\n" );
117       mutex_unlock_printfx();
118       later += interval;
119 #else
120       later += 90000; // 60 000 ns = 60 us
121 #endif
122       sleepUntil(later);
123     }
124 }
125 OS_TASKENTRY(task1_core3){
126 #ifndef SIM
127    mutex_lock_printfx();
128    printfx("This is MCSmartOS, starting...\n ======= \n Task1 on
          core3\n ======= \n\n");
129    mutex_unlock_printfx();
130 #endif
```

```
131
132     Time_t later;
133     Time_t interval = (Time_t)(getOneSecond() * 2);
134     while (1)
135     {
136         getCurrentTime(&later);
137         LEDtoggle(LED_05);
138
139 #ifndef SIM
140         mutex_lock_printfx();
141         printfx("Task1 from Core3 alive ;)\n" );
142         mutex_unlock_printfx();
143
144         later += interval;
145 #else
146         later += 80000; // 60 000 ns = 60 us
147 #endif
148         sleepUntil(later);
149      }
150 }
151
152 OS_TASKENTRY(task2_core3)
153 {
154 #ifndef SIM
155     mutex_lock_printfx();
156     printfx("This is MCSmartOS, starting...\n ======= \n Task2 on
            core3\n ======= \n\n");
157     mutex_unlock_printfx();
158 #endif
159
160     Time_t later;
161     Time_t interval = (Time_t)(getOneSecond() * 3);
162
163     while (1)
164     {
165
166         getCurrentTime(&later);
167         LEDtoggle(LED_06);
168 #ifndef SIM
169         mutex_lock_printfx();
170         printfx("Task2 from Core3 alive ;)\n" );
171         mutex_unlock_printfx();
172         later += interval;
173 #else
174         later += 90000; // 60 000 ns = 60 us
175 #endif
```

```
176         sleepUntil(later);
177     }
178 }
179
180 OS_TASKENTRY(task1_core4)
181 {
182 #ifndef SIM
183     mutex_lock_printfx();
184     printfx("This is MCSmartOS, starting...\n ======= \n Task1 on
            core4\n ======= \n\n");
185     mutex_unlock_printfx();
186 #endif
187     Time_t later;
188     Time_t interval = (Time_t)(getOneSecond() * 2);
189     while (1)
190     {
191         getCurrentTime(&later);
192         LEDtoggle(LED_07);
193 #ifndef SIM
194         mutex_lock_printfx();
195         printfx("Task1 from Core4 alive ;)\n" );
196         mutex_unlock_printfx();
197         later += interval;
198 #else
199         later += 80000; // 60 000 ns = 60 us
200 #endif
201           sleepUntil(later);
202      }
203 }
204
205 OS_TASKENTRY(task2_core4)
206 {
207 #ifndef SIM
208     mutex_lock_printfx();
209     printfx("This is MCSmartOS, starting...\n ======= \n Task2 on
            core4\n ======= \n\n");
210     mutex_unlock_printfx();
211 #endif
212
213     Time_t later;
214     Time_t interval = (Time_t)(getOneSecond() * 3);
215     while (1)
216     {
217         getCurrentTime(&later);
218         LEDtoggle(LED_08);
219 #ifndef SIM
```

```
220         later += interval;
221         mutex_lock_printfx();
222         printfx("Task2 from Core4 alive ;)\n" );
223         mutex_unlock_printfx();
224 #else
225       later += 90000; // 10 000 ns = 10 us
226 #endif
227       sleepUntil(later);
228     }
229 }
```

Listing 6.2: Testcode for mutex example of printfx with two task per core.

6.1.3 Mutex LED state example code

```
1 #include <kernel/smartos.h>
2 #include <drv/led.h>
3 #include <arch/mutex.h>
4 #include <libs/printfx.h>
5 #include <string.h>
6
7 OS_CREATE_MUTEX(leds);
8 OS_TASK_CONSTR_CORE1(core1_LED_State_00, 600, 16);
9 OS_TASK_CONSTR_CORE2(core2_LED_State_01, 600, 16);
10 OS_TASK_CONSTR_CORE3(core3_LED_State_10, 600, 16);
11 OS_TASK_CONSTR_CORE4(core4_LED_State_11, 600, 16);
12
13 OS_TASKENTRY(core1_LED_State_00)
14 {
15    Time_t later;
16
17 #ifdef SIM
18    Time_t interval = 50000;
19 #else
20    Time_t interval = (Time_t)((getOneSecond()));
21 #endif
22
23    while (1)
24    {
25       mutex_lock(&mutex_leds);
26       LEDsetState(LED_01,LED_OFF);
27       LEDsetState(LED_02,LED_OFF);
28       getCurrentTime(&later);
29       later += interval;
30       sleepUntil(later);
31       mutex_unlock(&mutex_leds);
32
33       getCurrentTime(&later);
34       later += interval;
35       sleepUntil(later);
36    }
37 }
38 /* Consumer with id 2 on Core 2 */
39 OS_TASKENTRY(core2_LED_State_01)
40 {
41    Time_t later;
42 #ifdef SIM
43    Time_t interval = 50000;
44 #else
```

```
45    Time_t interval = (Time_t)((getOneSecond()));
46 #endif
47
48
49    while (1)
50    {
51
52       mutex_lock(&mutex_leds);
53       LEDsetState(LED_01,LED_ON);
54       LEDsetState(LED_02,LED_OFF);
55       getCurrentTime(&later);
56       later += interval;
57       sleepUntil(later);
58       mutex_unlock(&mutex_leds);
59
60       getCurrentTime(&later);
61       later += interval;
62       sleepUntil(later);
63
64    }
65 }
66 OS_TASKENTRY(core3_LED_State_10)
67 {
68    Time_t later;
69 #ifdef SIM
70    Time_t interval = 50000;
71 #else
72    Time_t interval = (Time_t)((getOneSecond()));
73 #endif
74
75    while (1)
76    {
77       mutex_lock(&mutex_leds);
78       LEDsetState(LED_01,LED_OFF);
79       LEDsetState(LED_02,LED_ON);
80       getCurrentTime(&later);
81       later += interval;
82       sleepUntil(later);
83       mutex_unlock(&mutex_leds);
84       getCurrentTime(&later);
85       later += interval;
86       sleepUntil(later);
87    }
88 }
89
90 OS_TASKENTRY(core4_LED_State_11)
```

```
91 {
92    Time_t later;
93 #ifdef SIM
94    Time_t interval = 50000;
95 #else
96    Time_t interval = (Time_t)((getOneSecond()));
97 #endif
98    while (1)
99    {
100      mutex_lock(&mutex_leds);
101      LEDsetState(LED_01,LED_ON);
102      LEDsetState(LED_02,LED_ON);
103      getCurrentTime(&later);
104      later += interval;
105      sleepUntil(later);
106      mutex_unlock(&mutex_leds);
107
108      getCurrentTime(&later);
109      later += interval;
110      sleepUntil(later);
111   }
112 }
```

Listing 6.3: Testcode for mutex LED state measurement.

6.1.4 Wishbone master controller code

```
1  /* Converter between CV30PE core interface and Wishbone interface */
2
3  `default_nettype none
4
5  module wb_master_controller
6  (  input wire rst,
7     input wire  clk,
8     core_if.core2wb_port core_instr,
9     core_if.core2wb_port core_data,
10    wb_if_master.core2wb_port  wb_instr,
11    wb_if_master.core2wb_port  wb_data,
12    input wire [7:0] data_atop_i,
13    input wire [4:0] irq_ack_core_4_i,
14    output wire  irq_id_core_4_o
15 );
16    //internal used variables
17    logic data_rmw_mode_p,data_rmw_mode_n;
18    logic cyc_instr;
19    logic cyc_data;
20    /************************instruction signals for WB
         ********************/
21    //assigns for instruction signals
22    assign core_instr.gnt    = core_instr.req & ~wb_instr.stall_i &
         wb_instr.ack_i;
23    assign core_instr.rvalid = wb_instr.ack_i;
24    assign core_instr.err    = wb_instr.err_i;
25    assign core_instr.rdata  = wb_instr.dat_i;
26
27    assign wb_instr.adr_o      = core_instr.addr;
28    assign wb_instr.dat_o      = core_instr.wdata;
29    assign wb_instr.we_o       = core_instr.we;
30    assign wb_instr.tga_o      = core_instr.tga;
31    assign wb_instr.sel_o      = core_instr.be;
32
33
34    //generation of cycle signals for instructions
35    always_ff @(posedge clk or posedge rst)
36    begin
37     if (rst)
38       cyc_instr <= 1'b0;
39     else
40       if (core_instr.req && !wb_instr.ack_i && !wb_instr.err_i)
41        cyc_instr <= 1'b1;
42       else
```

```
43        cyc_instr <= 1'b0;
44      end
45      //assigns for instruction cycles
46      assign wb_instr.cyc_o = cyc_instr;
47      assign wb_instr.stb_o = cyc_instr;
48
49      /**********************data signals for WB ********************/
50      assign core_data.gnt    = core_data.req & ~wb_data.stall_i &
            wb_data.ack_i;
51      assign core_data.rvalid = wb_data.ack_i;
52      assign core_data.err    = wb_data.err_i;
53      assign core_data.rdata  = wb_data.dat_i;
54
55      assign wb_data.adr_o      = core_data.addr;
56      assign wb_data.dat_o      = core_data.wdata;
57      assign wb_data.we_o       = core_data.we;
58      assign wb_data.tga_o      = core_data.tga;
59      assign wb_data.sel_o      = core_data.be;
60      //checking for RMW mode
61      always_comb
62      begin
63        if(data_atop_i == 8'b00001010 && core_data.req)
64        begin
65          data_rmw_mode_n =  1'b1;
66        end
67        else
68        begin
69          data_rmw_mode_n = data_rmw_mode_p;
70        end
71      end
72      //generation of cycle signals for data
73      always_ff @(posedge clk or posedge rst)
74      begin
75        if (rst)
76        begin
77          cyc_data <= 1'b0;
78          data_rmw_mode_p <= 1'b0;
79        end
80        else
81        begin
82          cyc_data <= 1'b0;
83          data_rmw_mode_p <= data_rmw_mode_n;
84          if (core_data.req && !wb_data.ack_i && !wb_data.err_i &&
              data_atop_i ==  8'b00001010)
85          begin
86            cyc_data <= 1'b1;
```

```
87            data_rmw_mode_p <=  1'b1;
88          end
89          else
90          begin
91             if (core_data.req && !wb_data.ack_i && !wb_data.err_i)
92             begin
93                cyc_data <= 1'b1;
94                data_rmw_mode_p <= 1'b0;
95             end
96          end
97        end
98     end
99     //assigns for data cycles
100    assign wb_data.cyc_o = cyc_data || data_rmw_mode_p;
101    assign wb_data.stb_o = cyc_data;
102
103 endmodule
104 `resetall
```

Listing 6.4: System Verilog code for Wishbone (WB) master controller

6.1.5 Startup code Operating System (OS) code

```
1
2#include <cpu/riscv_ri5cy/riscv_ri5cyASM.h>
3
4  .section .text
5  .global  __os_stack_begin
6  __os_stack_begin:
7    .word __os_stack_begin
8
9  .section .text
10 .global  __os_heap_begin
11 __os_heap_begin:
12   .word __os_heap_begin
13
14
15 .section .startup_code_core1
16 .global _start_core1
17_start_core1:
18 // init kernel stack with __os_stack_begin (PC relative)
19 la      sp, __os_stack_begin_core1
20
21
22 // TODO [RG] gb is needed, but I dont know yet exactly how it works
23 // Initialize global pointer
24 la      gp, _gp_core1
25
26 // memory initialization is done here only
27 // for saving some code space. crt0 has C
28 // code commented out to do the same
29
30 // clear the bss segment
31 la      t0, __bss_start_core1
32 la      t1, __bss_end_core1
33 1:
34 REG_S   zero,0(t0)
35 addi    t0, t0, REG_SZ
36 bltu    t0, t1, 1b
37
38
39 // Copy .data section from ROM to RAM
40 la      t0, __data_size_core1
41 la      t1, __data_ROMstart_core1
42 la      t2, __data_RAMstart_core1
43 1:
44 beqz    t0, 2f
```

```
45  REG_L   t3, 0(t1)
46  REG_S   t3, 0(t2)
47  addi    t1, t1, REG_SZ
48  addi    t2, t2, REG_SZ
49  addi    t0, t0, -REG_SZ;
50  j       1b
51  2:
52
53  // Copy .sdata section from ROM to RAM
54  la      t0, __sdata_size_core1
55  la      t1, __sdata_ROMstart_core1
56  la      t2, __sdata_RAMstart_core1
57  1:
58  beqz  t0, 2f
59  REG_L t3, 0(t1)
60  REG_S t3, 0(t2)
61  addi  t1, t1, REG_SZ
62  addi  t2, t2, REG_SZ
63  addi  t0, t0, -REG_SZ;
64  j       1b
65  2:
66
67  li    tp, 0
68  li    a0, 0   // a0 = argc
69  li    a1, 0   // a1 = argv
70  li    a2, 0   // a2 = envp = NULL
71
72  // pmp config (allow all)
73  li  t0, 0xffffffff
74  li  t1, 0x000f
75  csrrw zero, 0x3A0, t1
76  csrrw zero, 0x3B0, t0
77
78  // interrupt configuration
79  csrrw zero, mstatus, 0xe
80  csrrsi zero, mie, 0xf
81  // set task prio to zero
82  csrw 0x800, zero
83
84
85  li t0, 0x1c0
86  csrrw zero, mtvec, t0
87  li t0, 0x100
88  csrrw zero, 0x005, t0 // utvec
89
90  // jump to crt0 (PC relative)
```

```
91  j       os_crt0
92  // jump to crt0 (absolute)
93  // lui    x3,%hi(os_crt0)
94  // jalr zero,x3,%lo(os_crt0)
95
96
97
98
99   .section .startup_code_2
100 .global _start_2
101_start_2:
102 // init kernel stack with __os_stack_core2_begin (PC relative)
103 la      sp, __os_stack_begin_core1
104
105 // TODO [RG] gb is needed, but I dont know yet exactly how it works
106 // Initialize global pointer
107 la      gp, _gp_core2
108
109 // memory initialization is done here only
110 // for saving some code space. crt0 has C
111 // code commented out to do the same
112
113 // clear the bss segment
114 la      t0, __bss_core2_start
115 la      t1, __bss_core2_end
116 1:
117 REG_S   zero,0(t0)
118 addi    t0, t0, REG_SZ
119 bltu    t0, t1, 1b
120
121
122 // Copy .data section from ROM to RAM
123 la      t0, __data_size_core2
124 la      t1, __data_ROMstart_core2
125 la      t2, __data_RAMstart_core2
126 1:
127 beqz    t0, 2f
128 REG_L   t3, 0(t1)
129 REG_S   t3, 0(t2)
130 addi    t1, t1, REG_SZ
131 addi    t2, t2, REG_SZ
132 addi    t0, t0, -REG_SZ;
133 j       1b
134 2:
135
136 // Copy .sdata section from ROM to RAM
```

```
137 la      t0, __sdata_size_core2
138 la      t1, __sdata_ROMstart_core2
139 la      t2, __sdata_RAMstart_core2
140 1:
141 beqz  t0, 2f
142 REG_L t3, 0(t1)
143 REG_S t3, 0(t2)
144 addi  t1, t1, REG_SZ
145 addi  t2, t2, REG_SZ
146 addi  t0, t0, -REG_SZ;
147 j       1b
148 2:
149
150 li    tp, 0
151 li    a0, 0   // a0 = argc
152 li    a1, 0   // a1 = argv
153 li    a2, 0   // a2 = envp = NULL
154
155 // pmp config (allow all)
156 li  t0, 0xffffffff
157 li  t1, 0x000f
158 csrrw zero, 0x3A0, t1
159 csrrw zero, 0x3B0, t0
160
161 // interrupt configuration
162 csrrw zero, mstatus, 0xe
163 csrrsi zero, mie, 0xf
164
165 // set task prio to zero
166 csrw 0x800, zero
167
168 li t0, 0x1c0
169 csrrw zero, mtvec, t0
170 li t0, 0x100
171 csrrw zero, 0x005, t0 // utvec
172
173 // jump to crt0 (PC relative)
174 j       os_crt0
```

Listing 6.5: Startup code in assembler for the OS

I'm always learning something. Learning never ends.

– Raymond Carver