HELMUT SCHMIDT
UNIVERSITÄT
Universität der Bundeswehr Hamburg

FPGA-Based System Virtual Machines

Von der Fakultät Elektrotechnik

**der Helmut-Schmidt-Universität/
Universität der Bundeswehr Hamburg**

**zur Erlangung des akademischen Grades
eines Doktor-Ingenieurs
genehmigte**

DISSERTATION

vorgelegt von

Diplom-Informatiker Michael Marcel Eckert

aus Frankenberg(Sa.)

Hamburg 2014

Gutachter:	Prof. Dr. Bernd Klauer
	Prof. Dr. Udo Zölzer
Vorsitzender der Prüfungskommission	Prof. Dr. Gerd Scholl
Tag der mündlichen Prüfung:	31.03.2014

Gedruckt mit freundlicher Unterstützung der HSU-Universität der Bundeswehr Hamburg.

Acknowledgments

This thesis is the result of my work at the Institute of Computer Engineering at Helmut-Schmidt-University/ University of the Federal Armed Forces Hamburg.

Firstly I am very grateful to Prof. Dr. Bernd Klauer, my Chair, for his support to successfully complete my dissertation. Secondly I would like to thank the remaining members of my dissertation committee Prof. Dr. Scholl and Prof. Dr. Zölzer.

Thirdly I would like to thank all my current and former colleagues at Helmut-Schmidt-University for their patience to discuss with me on all the topics related to my thesis: Dominik Meyer, Rene Schmitt, Oliver Seidel, Klaus Hildebrandt and Igor Podebrad.

I would also thank my students that developed a variety of device controllers available in the PRHS framework to improve it's general purpose usability.

Finally I would like to thank my wife Sandra, my daughter Lena and my son Paul. Without their support and love, this thesis would have never been finished.

Contents

1. Introduction **1**
- 1.1. Motivation . . . 1
- 1.2. Purpose of this Work . . . 3
- 1.3. Structure of this Work . . . 3

2. Configurable Computing **5**
- 2.1. Configurable Hardware . . . 5
  - 2.1.1. Memory - Look Up Table . . . 5
  - 2.1.2. Multiplexers . . . 6
  - 2.1.3. AND/OR Matrices . . . 7
  - 2.1.4. Simple Programmable Logic Devices . . . 8
  - 2.1.5. Complex Programmable Logic Devices . . . 9
  - 2.1.6. Programming Technologies . . . 10
- 2.2. FPGAs . . . 10
  - 2.2.1. CLB - Configurable Logic Blocks . . . 11
  - 2.2.2. I/O-Blocks . . . 11
  - 2.2.3. Interconnection System . . . 12
  - 2.2.4. Additional Elements . . . 12
  - 2.2.5. Dynamic and Partial Reconfiguration Capabilities . . . 12
- 2.3. Reconfigurable Logic in Computer Architectures . . . 13
  - 2.3.1. CPUs and Reconfigurable Logic . . . 13
  - 2.3.2. Devices and Reconfigurable Logic . . . 14
  - 2.3.3. Memories and Reconfigurable Logic . . . 15
  - 2.3.4. Static Islands beside Reconfigurable Logic . . . 16
  - 2.3.5. Extensions and Combinations . . . 16

3. Virtual Machines **17**
- 3.1. Process Virtual Machines . . . 18
  - 3.1.1. Operating Systems . . . 18
  - 3.1.2. Emulators and Dynamic Binary Translators . . . 18
  - 3.1.3. Same-ISA Binary Optimization . . . 19
  - 3.1.4. High-Level-Language VMs . . . 19
- 3.2. System Virtual Machines . . . 19
  - 3.2.1. Processor Virtualization . . . 20
  - 3.2.2. Virtual Memory Virtualization . . . 21
  - 3.2.3. Input/Output Virtualization . . . 22
  - 3.2.4. Multiprocessor Virtualization . . . 24

4. FPGAs and Virtualization **27**
4.1. Related Work . . . 27
4.1.1. FPGA-Ressource Virtualization . . . 27
4.1.2. JOP: A Java Optimized Processor . . . 27
4.1.3. SDVM - Self Distributing Virtual Machine. . . . 28
4.2. FPGAs as Dynamic Machine Instantiation Facility . . . 29
4.2.1. Number of Guest Systems . . . 30
4.2.2. Memory Issues . . . 30
4.2.3. Device Issues . . . 34
4.2.4. CPU and System related Questions . . . 40
4.3. Requirements on a Reconfigurable Logic Area . . . 42
4.4. Discussion of the Proposed Idea in Relation to Conventional System VMs . . . 42
4.5. The Need For a Testing Framework . . . 44

5. PRHS Framework - Hardware **47**
5.1. General Overview on PRHS Hardware . . . 47
5.2. PRHS Bus Definition . . . 48
5.2.1. PRHS Bus Interface . . . 49
5.2.2. PRHS Bus Protocol . . . 49
5.3. PRHSpA - PRHS Processor ARM ISA . . . 54
5.3.1. PRHScA - PRHS Core ARM ISA . . . 55
5.3.2. System Co-processor . . . 60
5.3.3. MMU - Memory Management Unit . . . 62
5.4. Base System Devices . . . 62
5.4.1. I/O Devices Fundamentals . . . 63
5.4.2. PRHS Bus Controller - Multiplexing primary and secondary PRHS Bus . . . 65
5.4.3. BusComponentStatus - Managing Device Discovering . . . 67
5.4.4. intchip4prhs - Interrupt Management Device . . . 68
5.4.5. timer4prhs - Timing Measurement in the PRHS Framework . 69
5.4.6. uart4prhs - Basic User Interaction . . . 71
5.4.7. bram4prhs - Block RAM based Memory . . . 72
5.5. PRHS SD Bus Subsystem . . . 73
5.5.1. PRHS SD Bus Definition . . . 74
5.5.2. PRHS Bus to PRHS SD Bus Cache . . . 78
5.5.3. PRHS SD Bus Interconnection System . . . 82
5.5.4. PRHS SD Bus Bridges . . . 86
5.6. Platform Specific Devices . . . 88
5.6.1. pstwo4prhs - Mouse and Keyboard Interfaces . . . 88
5.6.2. sysace4prhs - Compact Flash Card Controller . . . 89
5.6.3. v5emac4prhs - 10/100 Mbit Ethernet Controller for Virtex5 FPGAs . . . 91

5.7. Partial Reconfiguration Extension . . . 92
5.7.1. Reconfigurable Module . . . 92
5.7.2. reconfIF4prhs - Reconfigurable Module Control Interface . . . 93
5.7.3. Reconfiguration Guard - Address Space Separation and Sensitive Signals Gating . . . 94
5.7.4. icap4prhs - internal configuration access port . . . 94
5.8. Composed Hardware Modules . . . 95
5.8.1. base - Basic System . . . 95
5.8.2. baseReconf - Extend Basic System with a PR extension Module 97
5.8.3. baseReconfTop modules - Board Specific Top Modules . . . 97

6. PRHS Framework - L4PRHS 99
6.1. General Overview on L4PRHS . . . 99
6.2. Processor Specific Adaptations . . . 100
6.3. Machine Specific Adaptations . . . 103
6.4. PRHS Specific Device Drivers . . . 110
6.4.1. Overview on Linux Device Model . . . 110
6.4.2. Block Device Drivers . . . 111
6.4.3. Character Device Drivers . . . 112
6.4.4. Network Device Drivers . . . 114
6.4.5. Partial Reconfiguration Extension related Device Drivers . . . 115

7. PRHS Framework - Software 119
7.1. C Compiler Toolchain . . . 119
7.2. PRHS Bootstrapping . . . 121
7.2.1. Expected Hard Disk Structure . . . 121
7.2.2. First Stage Boot Loader . . . 122
7.2.3. Second Stage Boot Loader . . . 122
7.3. System Base Software . . . 123

8. Proof of Concept 125
8.1. Hardware . . . 125
8.1.1. Idle Configuration . . . 125
8.1.2. UART Configuration . . . 126
8.1.3. Core Configuration . . . 126
8.2. Device Drivers . . . 127
8.2.1. Idle Configuration . . . 127
8.2.2. UART Configuration . . . 127
8.2.3. Core Configuration . . . 127
8.3. Software . . . 128
8.3.1. Idle Configuration . . . 129
8.3.2. UART Configuration . . . 129
8.3.3. Core Configuration . . . 129
8.4. Proof of Concept in Comparison to Main Idea . . . 130
8.4.1. Processor related Results . . . 130

8.4.2. Memory related Results . . . 132
8.4.3. Device Sharing related Results . . . 132
8.4.4. Machine related Results . . . 133

9. Conclusion **135**
9.1. Summary . . . 135
9.2. Future Prospects . . . 136
9.2.1. Thoughts on the Pause, Suspend and Resume Problem . . . 136
9.2.2. Thoughts on FPGA Design Flows . . . 138
9.3. Application Areas for Reconfigurable Logic based System VMs . . . 138

Literature **145**

A. VHDL Coding Conventions **147**
A.1. Signal Naming Conventions . . . 147
A.2. Component Instantiation Conventions . . . 148

B. Devices - Platform Overview **149**

1. Introduction

1.1. Motivation

In 1965 Gordon E. Moore stated: "The complexity for minimum component costs has increased at a rate of roughly a factor of two per year."[Moo65] Over the decades this has been transformed to: "The number of transistors per area will double every 18-24 months". The well known *Moore's Law*. An also well known deduction is: "calculation performance doubles every 18-24 month".

The implication "calculation performance follows transistor density" was practically achieved by using transistors for a variety of additional hardware features to increase clock speeds of integrated circuits. This approach had been successfully used till the very first years of the 21st century. Today modern processor systems perform with clock speeds between one and up to four GHz. In Figure 1.1 the age before the millenium change is labeled as *Age of clock speed*. Due to different physical effects the clock speeds are facing the so called frequency wall.

To turn more and more transistors still delivered by the road maps of silicon foundries into performance, the total of cores has been increased on processor designs. This fact is considered as *Age of parallelism* in Figure 1.1. It is easy to predict that the curve of the core will also face its wall due to Amdahls law. Assuming this trend will be stable the next few years, a single chip might hold more than 256 processor cores in a few years. This might be beneficial for supercomputing needs, but for the general purpose computer it is oversized.

Configurable circuits are a promising approach to turn increasing transistor totals into performance. This is denoted in Figure 1.1 as *Age of configuration*. This requires significant changes in operating systems, in the areas of software and system engineering. Traditional approaches were focused on optimizing algorithms to perform on standard parallel hardware. The *configware* approach is vice versa in the sense that hardware is now optimized to support algorithms in form of *accelerator units*. Todays trend is to use configurable computing to add those accelerator units to existing computer architectures.

Hence, it is expected that future (configuration age) processors contain a proportionate amount of cores, whereas the remaining available transistors are used to form reconfigurable areas. In those areas, the accelerator units can be placed on an

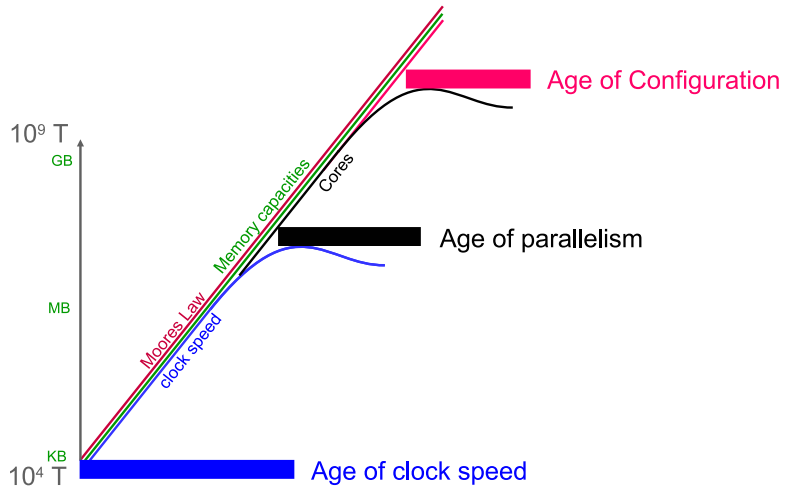


Figure 1.1.: Moore's Law and the consequences for computer architectures. Memory capacities are scaling directly with Moore's law. So did the clock speeds until the very early 2000s. Then physical effects limited the clock speeds to 1-4Ghz. [Kla13] expects the same for the total of cores for the mid range future. To take profit from a still increasing total of transistors configurable computing seems to be a promising technology. (Figure source: [Kla13])

as needed basis [Kla13].

Another area of interest in computer science are virtual machines (VMs). For desktop computers, system virtual machines received renewed interest during the last decade. Especially after x86 based processors of Intel and AMD were enabled to support native executed system virtual machines. A system virtual machine allows the execution of one or more guest operating systems while the original host operating system is still running. This approach has several advantages:

Robust systems: A failure of a guest machine doesn't affect the overall system.

Enhanced security: System virtual machines are often called sandboxes in the area of IT-security to emphasize the ability to separate the guest systems against each other.

Resource utilization: Instead of running multiple computers, which are busy most of the time, virtual machines provide a way to enhance the usage of one physical computer, presented as an illusion of being multiple, occasionally used ones.

Mixed OS platforms If the user of a computer system wants to use different software, only available for different operating systems, it's much easier to switch between several guest operating system instead of booting the necessary one every time it is needed.

Multi platform development For a software developer virtual machines provide two advantages: Firstly, a VM can be used to emulate different hardware configurations. The second advantage refers to the fact, that testing software on different (virtualized) operating systems is faster as switching between them by means of shutting one down and booting another one.

Migration Migrating to a new operating system can be tested step by step without the need to remove the old operating system from a machine or use an additional machine, only required for the migration period.

1.2. Purpose of this Work

This thesis presents the approach of taking advantage of reconfigurable logic by combining it with the idea of virtual machines. This combination allows to instantiate additional hardware resources for virtual machines, in opposition to the emulation approach of conventional virtual machines.

In this thesis, the requirements and dependencies among reconfigurable logic and operating systems are investigated to enable the use of virtual machine mechanisms on systems, combining hard-wired (static logic) and reconfigurable logic. To proof the feasibility of this idea, a proof of concept demonstrator is needed. To implement such a proof of concept demonstrator a testing framework, the Partial Reconfigurable Heterogeneous System (PRHS) framework has been developed. This framework is also presented in this thesis. Finally, the benefits of those reconfigurable logic based virtual machines are discussed in theory and in context of the proof of concept demonstrator.

1.3. Structure of this Work

This thesis is structured as follows:

In chapter 2 a brief introduction on (re)configurable computing will be given.

Conventional virtual machines will be briefly introduced in chapter 3. This is necessary to define a common terminology, as the term *virtualization* is extensively used in computer science in different senses. Additionally, this chapter is the basis for the theoretical and practical evaluation of the main idea of this thesis.

In chapter 4 the main idea of this thesis will be shown in detail. This includes the deduction of requirements for reconfigurable logic based virtual machines and also a discussion on the theoretical benefits.

In chapter 5, 6 and 7 the developed PRHS framework will be presented. Due to it's complexity, the hardware, operating system and software parts of the PRHS framework are presented in separate chapters.

The PRHS framework is used to implement a proof of concept demonstrator, as will be shown in chapter 8. This demonstrator is used to show the applicability of reconfigurable logic based system virtual machines.

Chapter 9 summarizes the results of this thesis and gives future prospects regarding the main idea of this thesis.

2. Configurable Computing

According to [CH02] *(re)configurable computing is intended to fill the gap between hardware and software, achieving potentially much higher performance than software, while maintaining a higher level of flexibility than hardware.* Therefore, an overview about the principle mechanisms of (re)configurable hardware is presented in this chapter. Focus lies on Field Programmable Gate Arrays (FPGAs) because they are todays most flexible reconfigurable devices and are in central scope of this thesis.

2.1. Configurable Hardware

Without elaborating too much details, each piece of hardware can be classified into storage elements, like Latches and Flip-Flops, as well as combinational circuits,like gates. By combining those elementary elements, larger circuits with more computational functionality can be implemented.

Configurable hardware summarizes those circuits, whose functionality can be changed after the circuits leave the silicon fabric.

2.1.1. Memory - Look Up Table

A boolean function is a transformation given by

$$\mathbb{B}^m \rightarrow \mathbb{B}^n; \mathbb{B} = \{0, 1\}; m, n \in \mathbb{N}$$

Each boolean function can be implemented by a dedicated combinational circuit.

A memory with an address width of m bits and a word width (number of bits per addressable memory cell) of n can implement each of the 2^{m*2^n} possible boolean functions of the form given in the above equation. This can easily be achieved by putting the truth table into the appropriate memory cells.

The following example illustrates this approach.

A boolean function of the form:

$$f(a, b) \rightarrow (w, x, y, z); a, b, w, x, y, z \in \mathbb{B} = \{0, 1\}$$

is given by the following truth table:

a	b	w	x	y	z
0	0	0	0	0	0
0	1	0	1	1	1
1	0	0	0	1	1
1	1	1	1	1	0

Figure 2.1 shows, how the truth table is mapped into the bits of the memory.

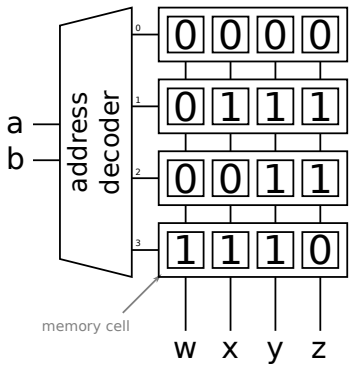


Figure 2.1.: Truth table mapped into memory.

The contents of a RAM can be altered at runtime and most of today ROMs are also (re)programmable (e.g. EEPROMs). Hence, a memory can be seen as the simplest form of a (user) (re)configurable hardware circuit.

2.1.2. Multiplexers

Look Up Table

The key concept of using a memory as a (re)configurable hardware device is, to put the truth table of the required boolean function into the memory cells. The memories address decoder is used to select the corresponding line of the truth table.

This principle can also be achieved by using a multiplexer, where the select inputs correspond to the address bits of the memory and the input lines are the corresponding bits of the truth table. As a multiplexer usually only has one output, it is necessary to use n multiplexers in parallel.

Regarding to the example of the previous section, the corresponding multiplexer solution can be found in Figure 2.2.

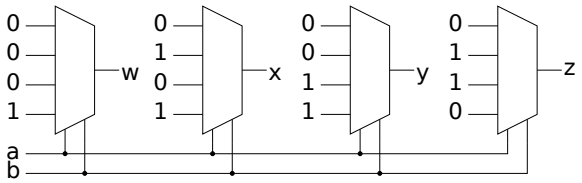


Figure 2.2.: Truth table implemented by multiplexers.

If the data inputs of the multiplexers are connected to some kind of storing element (e.g. a Flip-Flop or SRAM cell), this solution can be seen as another kind of (re)configurable circuit for implementing a boolean function.

Shannon Expansion

Another way of utilizing Multiplexers in reconfigurable logic is to take advantage of the Shannon expansion [Sha49] (in the following equation, expansion is about x_0):

$$f(x_0, x_1, \ldots, x_{n-1}) = x_0 \wedge f(1, x_1, \ldots, x_{n-1}) \vee \overline{x_0} \wedge f(0, x_1, \ldots, x_{n-1})$$

Applying the Shannon expansion n times for the above equation will result in the disjunctive normal form. This allows to implement the multiplexer look up table solution presented above.

If it is applied only $n-1$ times, the solution is still applicable to multiplexers. The remaining nth literal, that has not been a subject to the Shannon expansion is used as an input to the multiplexer. The other $n-1$ signals are used as the multiplexer select signals. This approach requires a programmable interconnection matrix, to select the right literals as multiplexer input and select signals.

For the above given example, the results for the solution are given in Figure 2.3.

2.1.3. AND/OR Matrices

According to [HM04], a programmable two-stage AND/OR matrix is the essential component of a Programmable Logic Device (PLD). Such an AND/OR matrix is suitable for implementing boolean functions given in disjunctive normal form. An example for such an AND/OR matrix, implementing the boolean function example of the previous section, is shown in Figure 2.4.

The AND matrix and the OR matrix can be either programmable or are already hard wired when the circuits are leaving the factory. Depending on the possible

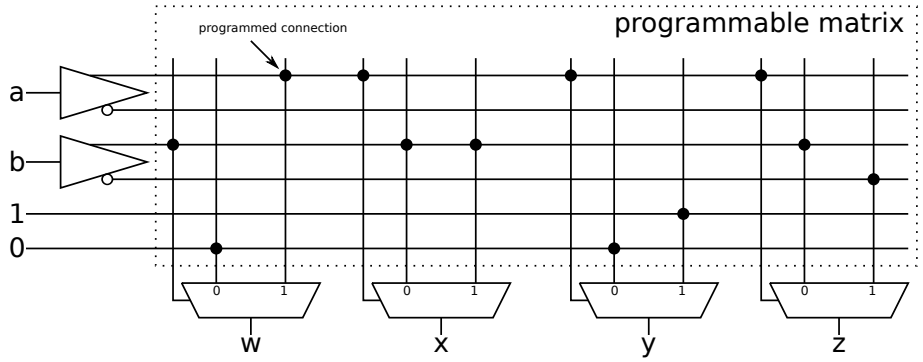


Figure 2.3.: Shannon expansion results implemented by multiplexers. Thereby, w is expanded about b; x, y and z are expanded about a.

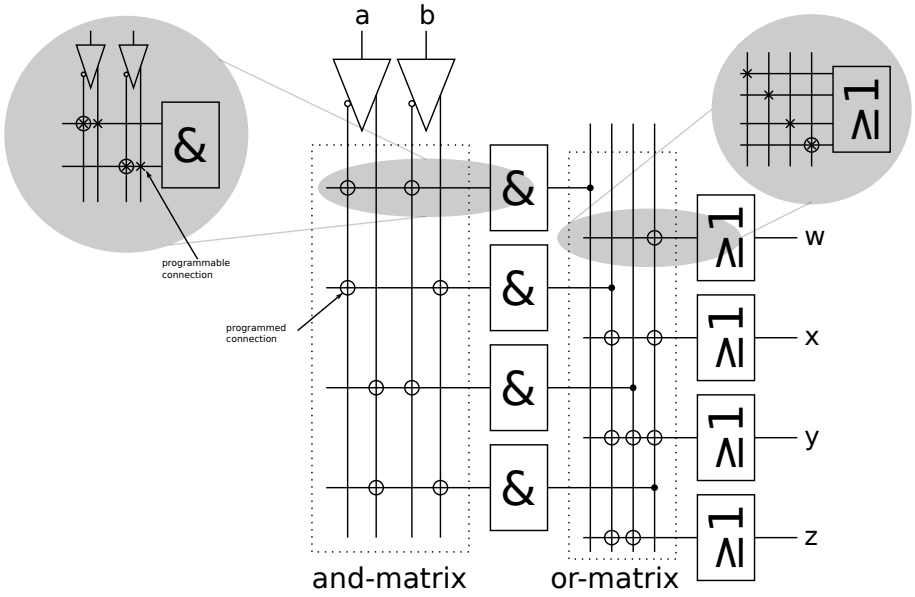


Figure 2.4.: Example for an AND/OR matrix implementation [HM04].

combinations of programmable/hard wired for the AND and OR matrices, there are three different types of configurable device classes as shown in table 2.1[1]. Interestingly, memories also fit into this categorization, when the address decoder is interpreted as AND matrix and the memory cells as OR matrix.

2.1.4. Simple Programmable Logic Devices

All the programmable devices mentioned in the previous sections are suitable for implementing boolean functions and therefore can replace dedicated combinational

[1]If neither AND nor OR matrix are programmable, the device isn't regarded as configurable at all

device class	PROM (Programmable ROM), RAM	PLA (Programmable Logic Array)	PAL (Programmable Array Logic)
AND-matrix	hard wired	programmable	programmable
OR-matrix	programmable	programmable	hard wired

Table 2.1.: AND/OR matrix categorization for configurable devices [HM04], [Sca01].

circuits. Still missing is the feature of a storing element like a latch or Flip-Flop to be able to implement finite state machines. In Figure 2.5 an AND/OR matrix is combined with a Flip Flop, several multiplexers and a tri-state gate, forming a so called (output logic) macro-cell (OLMC) of a Simple Programmable Logic Device (SPLD). In addition to the programmable AND-matrix, the pins (1), (2) and (3)

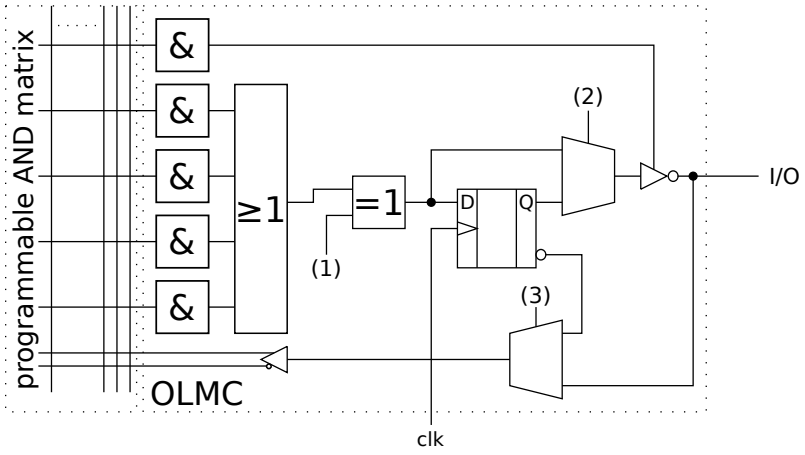


Figure 2.5.: An OLMC example (notional).

are also used for configuring the behavior of a macro cell.

The combination of several such OLMCs with an appropriately sized AND-matrix form a SPLD.

2.1.5. Complex Programmable Logic Devices

Connecting several SPLDs with an interconnection matrix and dedicated input/output stages on one chip results in a complex Programmable Logic Device (CPLD) as seen in Figure 2.6.

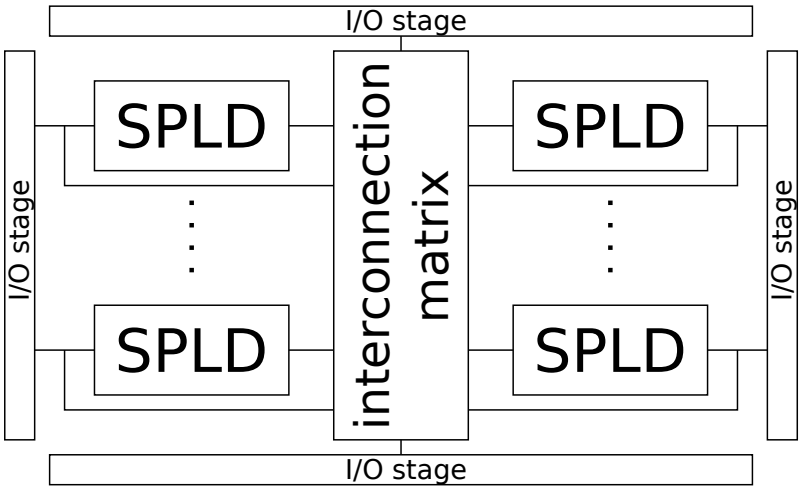


Figure 2.6.: A CPLD example [HM04].

2.1.6. Programming Technologies

There exist several different programming technologies for CPLDs and FPGAs. (FPGAs will be introduced in the next section.) The main distinguishing feature for the programming technology is reconfigurability. A programmable logic device is *one time configurable*, if it can be programmed only once, after it has left the silicon factory. If it is (re)programmable several times, it is called *reconfigurable*. For the reconfigurable programming technologies, there is an additional distinguishing feature: programming persistence. The programming can either be volatile, if the device has to be reprogrammed after every power down, or non-volatile, if the last programming is still available after a power down of the device. Categorization and corresponding programming technologies are summarized in the following table [HM04]:

category	one time configurable	non-volatile	volatile
technology	Fuse, Anti-Fuse	Flash, EEPROM	SRAM

2.2. FPGAs

FPGAs further develop the idea of the CPLDs. Both device classes have an amount of interconnected programmable blocks in common. For Field Programmable Gate Array (FPGA)s this amount is some orders of magnitude higher compared with CPLD. In addition the relation between boolean functions (LUTs) and storage elements (Flip Flops) is different for FPGAs and CPLDs. In general, FPGAs have more Flip-Flops in relation to the configurable combinational logic as CPLDs.

A FPGA consists of a number of three main elements:

CLB A configurable logic block is the basic element of a FPGA and contains the configurable logic.

I/O-blocks Input-/output-blocks connect the internals of a FPGA to the externals of the chip.

interconnection system The interconnections system connects the CLBs and the I/O blocks with each other. The interconnections are configurable.

The components will be described more detailed in the following subsections.

2.2.1. CLB - Configurable Logic Blocks

The basic element of a FPGA is a so called *Configurable Logic Block (CLB)*. The general structure of a CLB is shown in Figure 2.7.

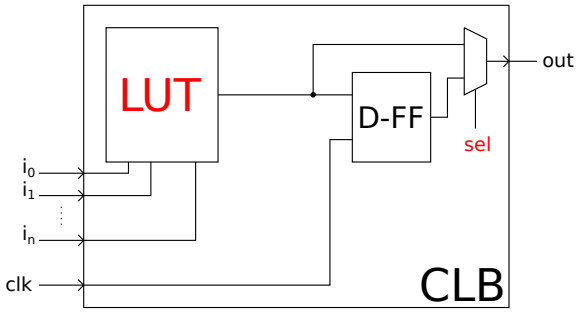


Figure 2.7.: General structure of a CLB.

A CLB consists of a Look-Up-Table (LUT) who's output might be used as an input to a Flop Flip or as direct output of the CLB. The size of a typical CLB-LUT ranges up to a 6-input LUT on todays FPGAs. The programmable components of the presented notional CLB are the LUT and the select-line of the output multiplexer (marked red in Figure 2.7).

2.2.2. I/O-Blocks

The Input-/Output blocks connect the pins of FPGAs with the internals of the chip. This provides the possibility to configure the pin as an input, as an output or as both of them (tristate, pull-up or pull-down). Additionally, it provides the possibility to wrap the Input/Output voltage levels and technologies (e.g. CMOS or TTL).

2.2.3. Interconnection System

The interconnection system connects the CLBs and I/O blocks of FPGAs. Depending on the technologies, there are three principle possibilities to organize FPGAs as shown in Figure 2.8.

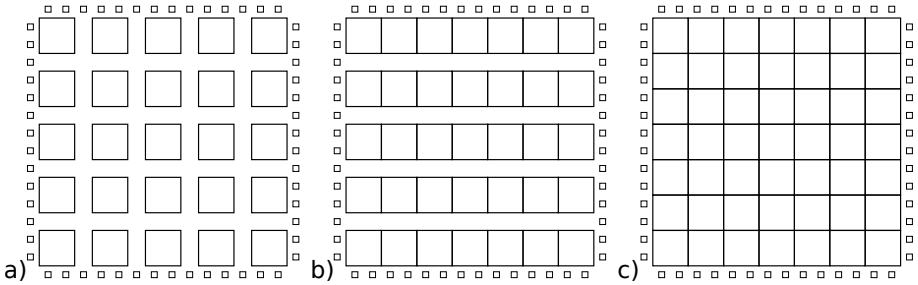


Figure 2.8.: Organizational structure of a FPGA, a) block oriented, b) line oriented, c) cell oriented [HM04].

block oriented Horizontal and vertical lines connect arrays of CLBs.

line oriented The CLBs are arranged as rows. Between those rows are the horizontal connections. Vertical connections are located in a layer above the CLBs.

cell oriented There is no CLB-interconnection in the CLB-layer. Horizontal and vertical connections are located in layers above the CLB-layer.

2.2.4. Additional Elements

In addition to the three main FPGA elements, presented in the previous sections, dedicated hardware components are possible. Common components, also available on todays FPGAs, are Block RAM, *Digital Signal Processors (DSPs)* and even entire processors (so called hard cores).

FPGA vendors also introduced chips, which include reconfigurable areas with an entire system on chip. (e.g. Zynq[Xil12b])

2.2.5. Dynamic and Partial Reconfiguration Capabilities

Early FPGAs and the corresponding work-flows only allowed to (re)configure the entire FPGA. As next development step, *partial reconfiguration* was developed, allowing only parts of the FPGA to be reconfigured, without touching the remaining parts.

Concerning those remaining parts, two possibilities arise, in how they are handled while the partial reconfiguration (PR) takes place. The *static partial reconfiguration* approach "freezes" the remaining parts of the FPGA, whereas the *dynamic partial reconfiguration* approach allows the remaining parts to keep running when the reconfiguration process takes place.

If the reconfiguration process is initiated and performed by the remaining part, a *in-system dynamic and partial reconfiguration* takes place.

2.3. Reconfigurable Logic in Computer Architectures

Besides the solely usage of a FPGA (e.g. for rapid prototyping), it is usually used within a computer architecture. Starting with the simplified system view of Figure 2.9 different possibilities exist, where to place reconfigurable logic and for what purpose to use it.

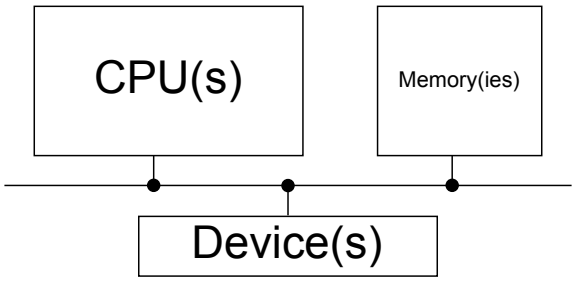


Figure 2.9.: A simplified system overview on a computer architecture.

Figure 2.10 gives a summarized overview on the different possibilities how to use reconfigurable logic in a computer architecture. Parts of a system, which are not reconfigurable at all are summarized as static hardware. Further discussion will be done in the subsequent sections.

2.3.1. CPUs and Reconfigurable Logic

This section is related to variant 1a and 1b of Figure 2.10. Variant 1a integrates reconfigurable logic into a processor itself. This results in the possibility to change the number of functional units (ALU, Multiplier, etc.) for super-scalar or VLIW architectures at runtime. Another possibility is to change the ISA of the processor itself at runtime. [Raz94], [NZ04] and [HK09] are academic examples for variant 1a).

Variant 1b provides mechanisms to reconfigure additional processing units (co-processors or processors) besides the static processors. The Convey HC-1 series

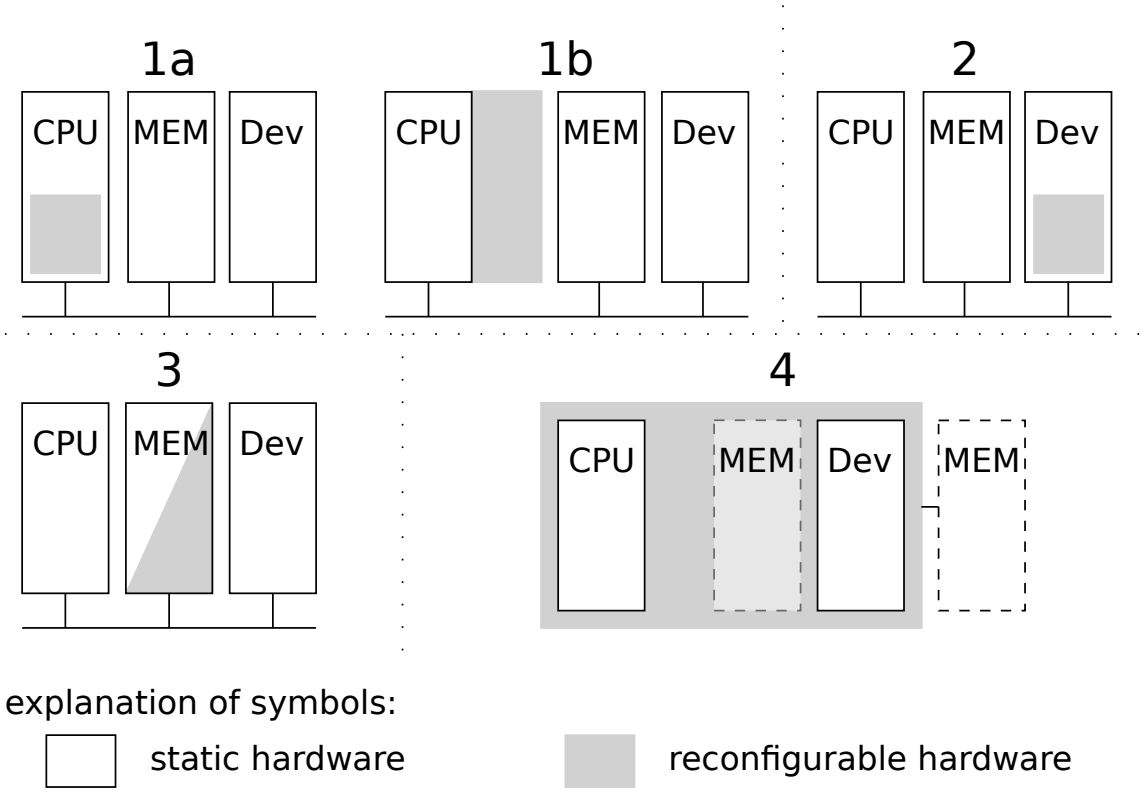


Figure 2.10.: Variants for including reconfigurable logic into a computer architecture.

(industrial), [HW97] and [Dal99] (academic) are examples, using the co-processor extension mechanism.

Architecture variant 1b is also referenced in literature as tightly coupled reconfigurable logic.

2.3.2. Devices and Reconfigurable Logic

This section is related to variant 2 of Figure 2.10. The term device isn't necessarily limited to I/O devices. Also function units, that are accessible as memory mapped or port mapped devices (in difference to variant 1b) are covered by this term. Convey HC-2 series (industrial) and several academic examples (e.g [HH09] or [PP04]) are based on this architecture variant.

This architecture variant is also referenced in literature as loosely coupled reconfigurable logic.

2.3.3. Memories and Reconfigurable Logic

This section is related to variant 3 of Figure 2.10. As CPU and devices have been investigated for their suitability to use reconfigurable logic among them, memories are investigated in this section.

In general two possibilities arise. The first one is to use reconfigurable logic to implement memory cells in it's pure sense. The second one is to provide reconfigurable logic besides the memory cells.

Reconfigurable Logic as Memory

CLBs, consisting of Flip-Flops and LUTs are presented as central element of FPGAs in section 2.2. The Flip-Flops can be used to form memory cells, the LUTs can be used to instantiate encoding and decoding logic. Hence, memory functionality can be configured into reconfigurable logic.

This approach is extremely resource intensive and even the most modern FPGAs can only form a size limited memory of just a few MByte.[2] For this reason the FPGA vendors started to include dedicated memories (Block RAM) into their FPGAs to get on chip memory at low area costs.[3] In summary, FPGAs are not suitable to instantiate large amounts of memory.

Reconfigurable Logic as Memory Supporting Elements

If a memory shall not only hold data, but implement some kind of intelligence, as it is for content addressable memory (CAM, also knwon as associative storage), another possibility to take advantage of reconfigurable logic exists. For a RAM, one ask for data at a given address. For a CAM, it's the other way around, one asks for the addresses, where specified data can be found. For this reason, CAMs are more complex than RAMs, as they have to include pattern matching and search algorithms in hardware. For a more detailed introduction on CAMs see [PS06].

It is imaginable to instantiate those pattern matching and search algorithms of CAMs in reconfigurable logic and therefore reduce the corresponding area consumption as those algorithms can be (re)configured as needed. The idea of implementing CAMs on FPGAs has been described and implemented by several authors like in [UKJCC12] or [GLD00].

[2] The current largest FPGA of Xilinx (XC7VX1140T) can form 2.2 MB of distributed RAM, if it is used entirely as memory. (See current FPGA family product overviews)

[3] The current largest FPGA of Xilinx (XC7VX1140T) contains 8.5 MB of Block RAM. (See current FPGA family product overviews)

2.3.4. Static Islands beside Reconfigurable Logic

This section is related to variant 4 of Figure 2.10. In opposite to variants 1 to 3, where reconfigurable logic is used to support static logic, variant 4 turns the principle upside down. On a reconfigurable chip, dedicated (not reconfigurable) hardware is added next to reconfigurable logic. Regarding processors, they are called hard-cores. Examples are Xilinx FPGAs including a PowerPC core.

Additionally, dedicated I/O device logic can be added to the reconfigurable chip. Zynq series of Xilinx [Xil12b] or Altera SoC-FPGAs [Cor13] are examples, where hard-cores and dedicated I/O device logic are integrated onto a chip, who's remaining area is used as reconfigurable logic.

As already discussed in the previous section, reconfigurable logic is not suitable to implement large amounts of memory. Therefore, these architectures usually provide dedicated external memory. The memory controller itself may reside on the FPGA.

2.3.5. Extensions and Combinations

In the previous subsections different approaches of combining reconfigurable logic with conventional computer architectures have been discussed. Combinations of those variants are also possible. The presented variants are all limited to single computers. Extension to multi-computer systems like clusters are also possible and have already been implemented, like the Cray XD1 ([cra04] and [UHT]) (industrial) or Axel Cluster [TL10] and RAMPSoC [GHSB08] (academic).

3. Virtual Machines

The concepts of virtualization and virtual machines are widely and extensively used mechanisms in computer science as they provide platform independence, effective resource sharing and security by isolation. They are also part of the main idea of this thesis.

Before starting a discussion about virtual machines, the term machine has to be defined. The meaning is dependent on the perspective.

From the perspective of a process, a machine consists of a logical memory address space, assigned to that machine, along with user level-registers (of the processor) and instructions that allow the execution of program code, associated with the process. I/O is only visible for a process through an operating system. Therefore, for a process a machine is constituted by the underlying hardware, the operating system and additional software providing necessary interfaces for the process (see Figure 3.1a).

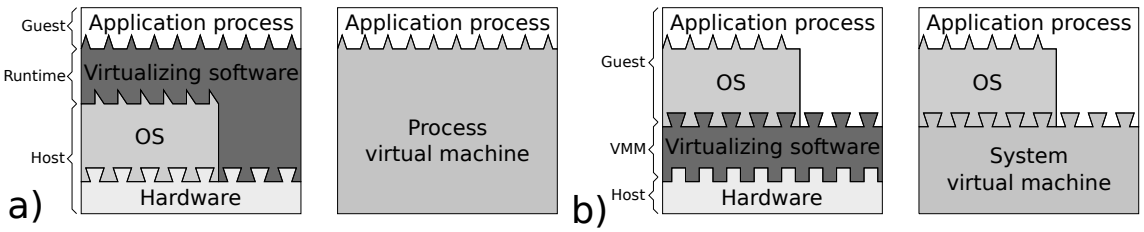


Figure 3.1.: Process and System virtual machines [SN05a].

From the perspective of an operating system, an entire system runs on an underlying machine. The system is an execution environment for applications (processes), managing the sharing of the available hardware resources between those applications. Hence, the machine is defined by the underlying hardware characteristics alone (see fig 3.1b).

For these given reasons, there are two types of virtual machines in general: process virtual machines and system virtual machines. A process VM exists to execute a process. The (process) machine is constituted by hardware, operating system and an optional virtualization software. A system VM provides a complete environment to execute a (guest) operating system. The system machine is constituted by hardware and a virtualization software.

The virtualizing software, that implements a process VM, is often termed *runtime software*. The virtualizing software in a system VM is typically referred to as the *Virtual Machine Monitor* (VMM).

This chapter is based on verbatim excerpts and summaries of [SN05b] and [SN05a].[1]

3.1. Process Virtual Machines

Process VMs provide a virtual programming interface (application programming interface (API) and standard libraries) for user applications.

3.1.1. Operating Systems

Operating Systems (OS) are the oldest form of process virtual machines. They give each process the illusion of having a complete machine for their own. The OS supports this illusion by managing resources (processor(s), devices, memory) and enforce time-sharing of theses resources among the different processes. In consequence an operating system provides a replication of the underlying hardware for each concurrently executed program. If the operating system is regarded as process virtual machines host, there is no need for a dedicated virtualization software. The OS itself is the virtualization software in this case.

3.1.2. Emulators and Dynamic Binary Translators

A challenging problem occurs, if a program shall be executed on machine that has a different ISA than the intended ISA, this program was originally compiled for. The ISA of the program has to be *emulated* by the process executing environment.

Interpretation is a straightforward way to implement this emulation. The interpreter program fetches, decodes, and emulates (almost each single) instruction of the guest ISA. This process is very slow, as emulating one instruction of the guest ISA might require several host ISA instruction.

Dynamic binary translation is an approach to speed up the translation process by not translating the guest program instruction by instruction, but on the basis of blocks of consecutive instructions. By saving already translated blocks, and re-using them if necessary, the relatively high overhead of translation can be reduced.

[1] [SN05a] is a summarizing article of [SN05b].

3.1.3. Same-ISA Binary Optimization

Dynamic binary translation, as described above, can include code optimization to reduce performance loss. This capability enables to implement VMs, wherein the guest and host ISA are the same, with optimization of a program as a VM's sole purpose. Same ISA dynamic binary optimizers use profile information collected during the interpretation or translation phase to optimize the binary executable on-the-fly.

3.1.4. High-Level-Language VMs

A key object for process VMs is cross-platform portability. If $m \in \mathbb{N}$ programs have to be used on $n \in \mathbb{N}$ different (operating) systems, several problems occur:

- Different operating systems use different system calls and application programming interfaces (APIs). Therefore, the sources have to be prepared for compiling them for different operating systems.
- $m \times n$ compilation steps are necessary to compile each of the m programs for all n operating systems.

To solve these time consuming problems, high-level-language VMs have been introduced. In summary a new intermediate application programming interface (API) is defined which is used by programs. To allow those programs to be executed on the different operating systems, a *run-rime-environment* is introduced. The run time environment has to be compiled for each of the m targeted operating systems. This is usually done by the run time environment developer, not necessarily the software developers. Each of the n programs have to be compiled for the run time environment without the necessity to differentiate between a lot of APIs.

Examples for Run Time Environments are the Java Virtual Machine or the .net Framework Run Time Environment.

3.2. System Virtual Machines

In a system VM, the VMM primarily provides platform replication. The central issue is dividing a set of hardware resources among multiple guest OS environments. The VMM manages and has access to all the hardware resources. A guest OS and it's application processes are then managed under hidden control of the VMM.

For this reason, the relationship between the VMM and a guest OS is analogous to the relationship between an operating system and an application in a conventional

system. In the latter, the OS typically works in a privilege level higher than the one of the applications (e.g. system-mode vs. user-mode of the processor), as shown in Figure 3.2 b). A virtual machine system, in which the VMM operates in a privilege

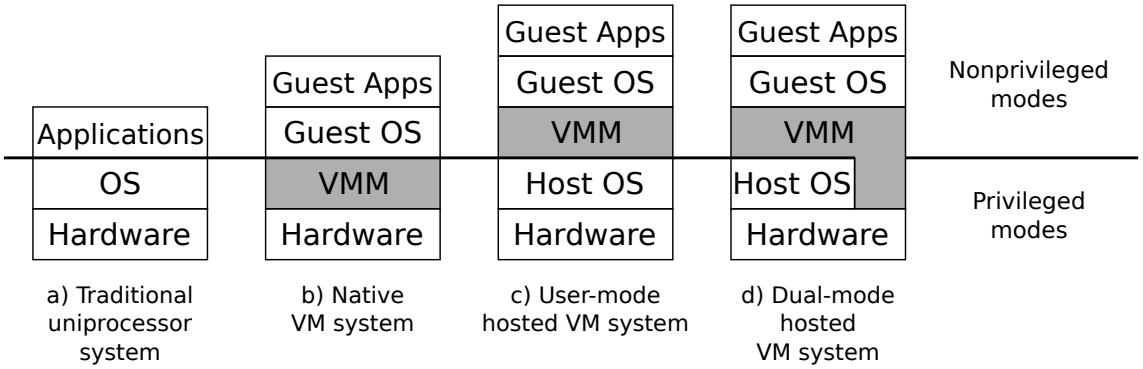


Figure 3.2.: Native and Hosted VM systems [SN05b].

mode higher than the mode of the guest virtual machine, is called *native VM system*. In literature, native VM systems and especially the VMM are also known as *Type 1 Hypervisor* (see [Tan07]).

In a *host VM system* the VMM is built upon the already available functionalities of a host operating system. Whether or not the VMM is allowed to run even parts of it (e.g. in form of device drivers) in a more privileged mode, there's a differentiation into *user-mode hosted VM systems* and *dual-mode hosted VM systems* as given in Figure 3.2 c) and d). Which one to choose mainly depends on the possibility to modify host OS functionality. In literature hosted VM systems and especially the VMM are also known as either *Type 2 Hypervisor* or *Paravirtualization* depending on the applied virtualization techniques (see [Tan07]).

Which system VM type to choose, isn't a question of preference. The choice mainly depends on the "features" of the used hardware and especially the used ISA which itself is predominated by the processor. Hence, different virtualization aspects related to processor, memory and devices will be discussed in the following sections. These aspects are important for the later discussion of the main idea of this thesis.

3.2.1. Processor Virtualization

The key aspect of virtualizing a processor lies in the execution of the guest operating system instructions including both, system-level and user-level instructions. As can be easily seen in Figure 3.2, the guest operating system runs entirely in non-privileged mode, regardless if it is a native or a hosted VM. Therefore, system-level instructions (also known as *sensitive instruction*) are of special interest, as the guest OS can't execute these, but needs to have the perception of doing so.

There are two general ways to virtualize a processor:

Emulation The methods emulation and dynamic binary translation have already been presented in the process VM section. However, they are the only applicable methods, if the needed guest ISA is different to the ISA implemented by the hardware.

Even if host and guest ISA would be the same, emulation and dynamic binary translation can be used to replace the sensitive instruction of the guest system with special calls to the VMM, emulating or interpreting these instructions. This is the typical use case for hosted VM systems.

For Type-2 Hypervisors, the replacement is done at runtime. If the replacement of sensitive instructions isn't done at runtime of the guest system, but at compile time, the technique is called *paravirtualization*. It is only applicable, if the source code of the guest operating system is accessible and modifiable.

Direct Native Execution If host and guest ISA are the same, it is imaginable to execute the guest OS directly "as is" on the processor running in user mode. However, the problem of dealing with the guest OS's sensitive instruction remains.

Popek and Goldberg ([PG74]) stated, that direct native execution is only possible, when the set of sensitive instructions is a subset of the *privileged instructions*. A privileged instruction is an instruction, that causes a trap if executed in non-privileged mode. Under this assumption, each natively executed sensitive guest instruction will trap to the VMM. The VMM will then emulate the guests sensitive instruction, providing the guest OS the perception of running in a privileged mode.

This "simple" requirement for virtualizing a processors ISA was not fullfilled for general purpose x86 based desktop computers till 2005, when AMD introduced SVM (Secure Virtual Machine) and Intel VT (virtualization technology) [Tan07]. Till this time some sensitive instruction had been just ignored by the processor, when executed in user-mode. Direct native execution is used by Type-1 Hypervisors.

3.2.2. Virtual Memory Virtualization

In a system VM environment, each of the guest VMs has its own set of virtual memory tables. Address translation in each of the guest VMs transforms addresses in its virtual address space to locations in real memory – this real memory would correspond to the physical memory on a native platform; in a VM platform, however, this is not the case. Rather, in a system VM environment, a guest's real memory address has to undergo a further mapping to determine the address in physical

memory on the host hardware. Note that there is a clear distinction between real memory and physical memory – terms often used interchangeable in other contexts. Physical memory is the hardware memory. Real memory is a guests VMs illusion of physical memory; an illusion supported by the VMM, when it maps a guest's real memory to physical memory. It is this real to physical mapping that implements the virtualization of memory in a VM system. ([SN05b] section 8.3.1)

Regardless the type of system VM, native or hosted VMM, there are two methods of virtualizing the virtual memory of a guest OS, which depend on the way the corresponding Memory Management Unit (MMU) enforces the implementation of virtual memory:

Virtualizing Architected Page Tables If the architecture of a page table is defined by the ISA, and the operating system and underlying hardware (MMU) cooperate in maintaining and using it, we talk about architected page tables. In this case, the TLB is maintained only by the hardware and not visible to the OS. In the case of a TLB miss, the MMU "walks" the page tables to get the appropriate TLB entry, or signal a permission or page fault to the OS.

Virtualizing architected page tables is done by implementing so called *shadow page tables*. Accesses to the page table areas of main memory are sensitive instructions. Hence, each access to a page table has to trap to the VMM (regardless if hosted or native). The VMM has to update the "real" page tables appropriately. See [SN05b] section 8.3.2 or [Tan07] section 8.3.5 for details.

Virtualizing Architected TLBs On an architected TLB, the TLB is directly managed by the operating system. This TLB has to be virtualized now. The VMM has to hold a virtual TLB for each guest and also manage the "real" TLB. So each TLB access of a guest OS is sensitive now and has to be handled by the VMM (regardless if hosted or native) to keep the virtual TLB copies up-to date and set the real TLB correctly. See [SN05b] section 8.3.3 for details.

3.2.3. Input/Output Virtualization

The proliferation of I/O devices is also a problem for conventional operating systems, which have developed abstractions to support a wide variety of devices and device types. It is possible to adapt many of those techniques for use in system virtual machines.

The virtualization strategy for a I/O device consists of

1. constructing a virtual version of the device and then
2. virtualizing the I/O activity directed at that device.

When a guest system requests to use the virtual device, the request is intercepted by the VMM. The VMM converts the request to a request of the underlying physical device before it is finally carried out.

Virtualizing Devices

The technique used for virtualizing a device depends on whether it is shared and, if so, the way in which it can be shared. Following are the common categories for devices (see [SN05b] section 8.4.1 for more details):

Dedicated Devices Some I/O devices, by their very nature, must be dedicated to a particular guest or at least switched from guest to host on a very long time scale. Examples of dedicated I/O devices are the display, keyboard, mouse and speakers. However, a guest VM request to such a device will trap to the VMM, which can then issue the request. Interrupts of the device will always trap to the VMM, which can hand them over to the dedicated guest VM.

Partitioned Devices For some devices, such as a disk, it is convenient to partition the available resources among the virtual machines. A very large disk, for example, can be partitioned into several smaller virtual disks that are made available to a virtual machine as dedicated device. To emulate an I/O request for the virtual device such as a disk, the VMM has to translate the parameters for the underlying physical device.

Shared Devices Some devices, such as a network adapter, can be shared among a number of guest VMs at a fine time granularity. A request by a guest VM to use the device is translated by the VMM to a request for the physical device through a virtual device driver.

Spooled Devices A spooled device is shared, but at a much higher granularity than a device such as a network adapter. An example of a device that is often spooled is a printer.

Nonexistent Physical Devices Virtualization makes it possible to provide support for virtual devices "attached" to a virtual machine for which there is no corresponding physical device. The device itself is emulated by the VMM.

Virtualizing I/O Activity

On conventional (non-virtualization) operating systems, I/O abstraction is layered, resulting in different activities at the different layers. Therefore, the following levels, where I/O vitualization can take place, exist.

Virtualizing at the I/O Operation Level The privileged nature of I/O operations makes them easy for the VMM to intercept, because they trap in user mode. However, once intercepted, it may be difficult for the VMM to determine exactly, what I/O action is being requested. The VMM has to "reverse engineer" the multiple issued requests and deduce the complete I/O action a guest VM wants to perform.

Virtualizing at the Device Driver Level If the VMM can intercept the call to the virtual device driver, it can convert the virtual device information to the corresponding physical device and redirect the call to a driver program for the physical device. This scheme requires the VMM to have knowledge about the guest operating systems internal device driver interfaces.

Virtualizing at the System Call Level This approach brings the previous one to the boil. Instead intercepting device driver calls, I/O activity may be already intercepted at system call level. The VMM would need system call routines, that shadow the system call routines available to the user (of a guest VM). This is again, a very guest OS specific task, which requires more guest OS internals understanding than just device drivers.

3.2.4. Multiprocessor Virtualization

The previous system VM discussion focuses on uniprocessor system. Another interesting area, where system virtual machines are of interest are large shared-memory multiprocessors (SMP). Here, an important objective is to partition the large system into multiple smaller (multi- or even uni-)processor systems by distributing the hardware resources, available in the SMP system. Two general possibilities for partitioning can be found:

Physical partitioning The physical resources of one VM are disjoint from the resources of the others. This provides a high degree of isolation.

Logical partitioning The underlying hardware resources are time-multiplexed between different partitions and the associated virtual machines. Compared to physical partitioning, resource utilization is enhanced at the cost of hardware isolation benefits.

Both partitioning techniques require software controlled hardware support to enforce the partitioning.

4. FPGAs and Virtualization

In the previous chapters, (re)configurable computing and virtual machines are presented separately. In this chapter both ideas will be combined. First of all, related work will be discussed. Finally, the main idea of this thesis will be presented, occurring problems will be deduced and theoretical solutions for those problems will be shown as well.

4.1. Related Work

This section is dedicated to related work in the area of combining virtualization and configurable logic. For better understanding the differences of the systems and to draw a clear distinction to the main idea of this thesis, it is necessary to emphasize the different interpretations and meanings of the words *virtualization* and *(re)configuration* among the different authors.

4.1.1. FPGA-Ressource Virtualization

A lot of publications (e.g. [HH09] [PP04]) were issued on using a FPGA as an additional hardware resource, such as general purpose I/O resources, but for acceleration purposes. This technique is called virtualization because a device is virtualized by an operating system (device driver), but has to be considered separately from virtual machines.

Another area of interest for researchers is the *virtualization* of the FPGA itself. A common idea in this field is to introduce an abstraction layer between the hardware description and the targeted FPGAs to make the FPGA design flow more flexible, faster and device independent. Examples are [FP98], [HSE+00] and [MK11].

4.1.2. JOP: A Java Optimized Processor

This section is based on [SKKR11] and [Sch03]. JOP is a hardware implementation of the Java Virtual Machine (JVM) targeted for small embedded systems with real-

time constraints. It is implemented as a soft core for usage in a FPGA. The main features of JOP are as follows:

1. Fast execution of Java bytecode[1] without the need for an Just-In-Time compiler.
2. Predictable execution time of Java bytecode. Small core, that fits in a low cost FPGA.
3. Configurable resource usage through HW/SW co-design.
4. Flexibility for embedded systems through FPGA implementation.

Due to the variation in complexity of Java bytecode, not every JVM instruction can be implemented in hardware (e.g. *new* or *invokestatic*). JOPs solution for this problem is to execute only a subset of the bytecode native (on hardware) and trap more complex ones. These complex instructions are then handled by micro programs .

For the JOP project FPGAs have been used to implement a Java processor. It does not use further (re)configuration capabilities of FPGAs as it will be proposed by the main idea of this work. As Java is a process virtual machine (see previous section), the JOP approach is different to the system virtual machine approach of this thesis.

4.1.3. SDVM - Self Distributing Virtual Machine.

The self distributing virtual machine (SDVM) is an adaptive, self configuring and self distributing virtual machine for clusters of heterogeneous, dynamic computing resources [HEW05]. It was designed to feature undisturbed parallel computation while adding and removing processing units from computing clusters by implementing a middleware operating system.

This OS provides a unified view of an application onto the underlying changing hardware environment. The SDVM is therefore to be classified as a process virtualization platform. This is different to the main idea of this thesis, where system VMs are used.

In [HW08] the authors of the SDVM present the usage of FPGAs to implement a changing hardware environment providing additional capabilities to SDVM. Therefore, it is named SDVM^r when used with FPGAs. The reconfigurable logic areas are used in two ways within SDVM^r:

- Adopt the available degree of parallelism (number of *processing elements*), according to the needs of an application and the available FPGA resources.

[1] The instructions of the JVM

The processing elements are a combination of a processor, Block RAM, a timer and an interrupt controller. Such a single, reconfigurable logic based processing element of SDVM^r is not capable of running a full fledged operating system as it doesn't provide enough memory. This is an important difference to the proposed main idea of this thesis, as will be presented in the remainder of this chapter.

- Extent already available processing units with special functions (accelerator units). This is just another FPGA resource virtualization method.

4.2. FPGAs as Dynamic Machine Instantiation Facility

In this section, the main idea of this thesis is discussed. The main idea is to:

Exploit the possibilities of reconfigurable logic to instantiate hardware supported system virtual machines.

This includes the following paradigms:

1. Reconfigurable logic is used on purpose and by principle to instantiate entire machines with the option to run a full fledged operating system on each of those machines. Those instantiated systems can be seen as a guest system in terms of system virtual machine concepts. (instantiation paradigm)

2. All dedicated resources of the overall system are managed by the host operating system (Virtual Machine Manager), as is done by conventional virtual machine managers. This is an essential requirement to see the overall system as virtualization system. (virtualization paradigm)

If the first paradigm is not full filled, a conventional virtual machine is brought onto a FPGA. If the second paradigm is not full filled, two separate and independent systems reside on one FPGA; the only benefit would be the instantiation of one system by the other.

Figure 4.1 gives a simplified overview on the presented idea:

The arising questions, problems and the corresponding possible solution are discussed in the following.

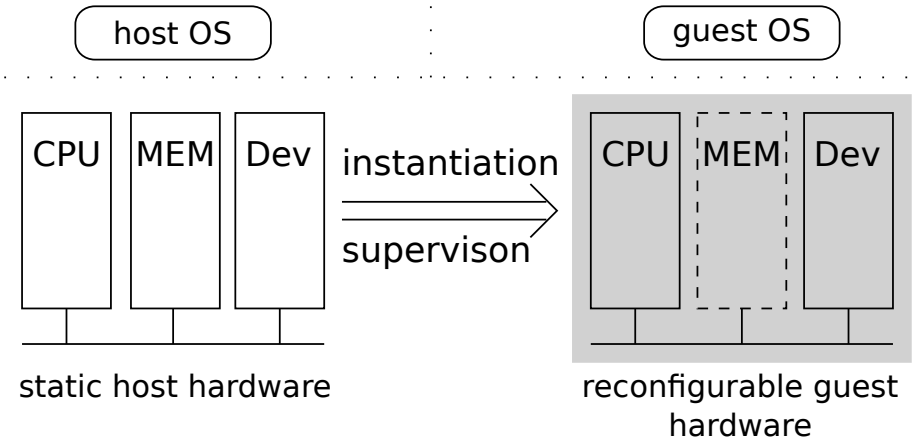


Figure 4.1.: Simplified overview on the main idea.

4.2.1. Number of Guest Systems

The number of guest systems is generally not limited. Limitations arise from the size of the available reconfigurable logic area and the area required for instantiating a guest machine.

Additionally, the formulated idea is recursive by principle. Recursion depth also depends on the available reconfigurable logic area.

4.2.2. Memory Issues

Concerning the memory of the guest machine, some questions/problems arise:

1. How much memory is needed by the guest system? Can this amount of memory be provided by the reconfigurable area?
2. How is the virtualization paradigm of the main idea (all resources are managed by the host operating system) enforced for the guest's memory?

These questions and possible solutions are discussed in detail in the following.

Providing enough Physical Memory for the Guest System

In section 2.3.3 the instantiation of memory in reconfigurable areas has been discussed. It has been stated, that it isn't reasonable to use reconfigurable logic to instantiate memory, but use Block RAM, also provided by todays FPGAs. If the Block RAM (or to be more precise, the resulting available amount of memory), usable in the reconfigurable area is sufficient for the guest system, no additional

memory resources have to be provided externally to the reconfigurable area.

For most embedded operating systems this assumption might hold. For full fledged desktop or server operating system the few MBytes of Block RAM, embedded in FPGAs, will not be sufficient to even hold the operating system in memory solely.

Therefore, it is recommended to provide the possibility of accessing the overall systems main memory to the guest system. From a traditional computer architectures perspective, this can be regarded as a dedicated DMA channel of the reconfigurable area.

A single shared memory controller can be a performance bottleneck for the host and several instantiated guest machines. Hence, it would be beneficial if the overall system provides more than one memory controller. The specific number is an engineering trade-off depending on the expected average number of guest systems running simultaneously.

Relation between the Host Operating System and Guest Memory

After the guest hardware is instantiated, the guest operating system needs to be started. Before it can be started, it has to be loaded to the appropriate position in the guest systems memory. This can either be accomplished by the host operating system or some kind of guest boot-loader executed on the guest processor.

Nevertheless, guest OS or boot-loader have to get into the guest machines memory, controlled by the host operating system (to fulfill the virtualization paradigm). For this reason a connection between the host hardware and the guest memory is necessary to enable the host system to load software into the guests memory.

1. In the case of a shared main memory, this connection exists by design.
2. In the case of Block RAM based memory, embedded in the reconfigurable area, a dedicated connection between host system bus and this memory has to exist.

Guest Virtual Memory Virtualization

Up to now memory in it's physical hardware representation has been investigated. Operating systems support virtualization of physical memory for giving an application the perception of having it's own memory. This virtual memory concept has to be supported by both - guest and host operating system.

Based on the above given arguments, a memory, accessible by both - the guest and host system is assumed in the following. It doesn't matter if this is the overall

system memory or a memory, embedded in the reconfigurable logic area(s).[2]

Approaches to virtualize virtual memory are constituted in section 3.2.2 including the differentiation of virtual, real and physical memory. On conventional system virtual machines, the real to physical mapping for the guest VM is enforced by the same Memory Management Unit(s), as host OS and guest OS are executed on the same processor(s). (See [SN05b] chapter 8.3. for further details.) As opposed to this, on virtualization systems, following the main idea of this thesis, host OS and guest OS run on different processors (as a consequence of paradigm 1). Hence, other mechanisms to implement real to physical memory mapping for guest systems are necessary. To support this mapping, additional hardware, the *Guest Memory Management Unit (GMMU)* is introduced, as shown in Figure 4.2.2.

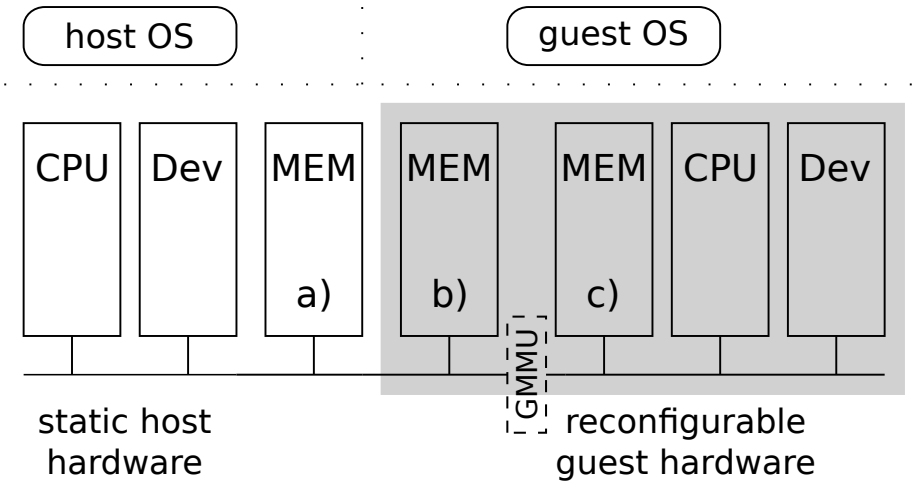


Figure 4.2.: Introducing a Guest Memory Management Unit (GMMU).

The memory can be the physical main memory of the overall system (a) or a memory instantiated in the reconfigurable area itself (b). Memory c) is not accessible by the host system, because it's behind the GMMU from the host's perspective. Therefore, memory c) is not subject of virtual memory virtualization.

The newly introduced GMMU can either be given in the static hardware part of the overall system (only if no reconfigurable logic based, host accessible memory is used; Figure 4.2.2 b)), or be instantiated in reconfigurable logic. If instantiated in static logic, the used interface and protocol for the guest machine's system bus is fixed. If the GMMU is instantiated in reconfigurable logic, the guest machine's system bus isn't fixed at all, providing more flexibility for the guest machine's characteristics.

The implementation complexity depends on the way, the real memory of the guest system is constituted in physical memory. For the further discussion, the following

[2]The discussion is presented with one guest system. The presented solutions would also apply to multiple guests.

assumptions are made to rely on the essentials of the proposed solutions:

1. The real and the physical address space for the memory of a system is contiguous.

2. The real and the physical address space for the memory of a system starts at address 0.

Both assumption can easily be resolved. The latter one by introducing an offset for the start address. The former one by replicating the presented solutions for each separate memory area.

In general, there are two possibilities to solve the problem of translating a guest's real addresses to the physically associated ones, as shown in Figure 4.3 and further explained in the following. Both solutions are adapted solutions of the real to physical mapping problem for multiprocessor virtualization as presented in [SN05b] chapter 9.

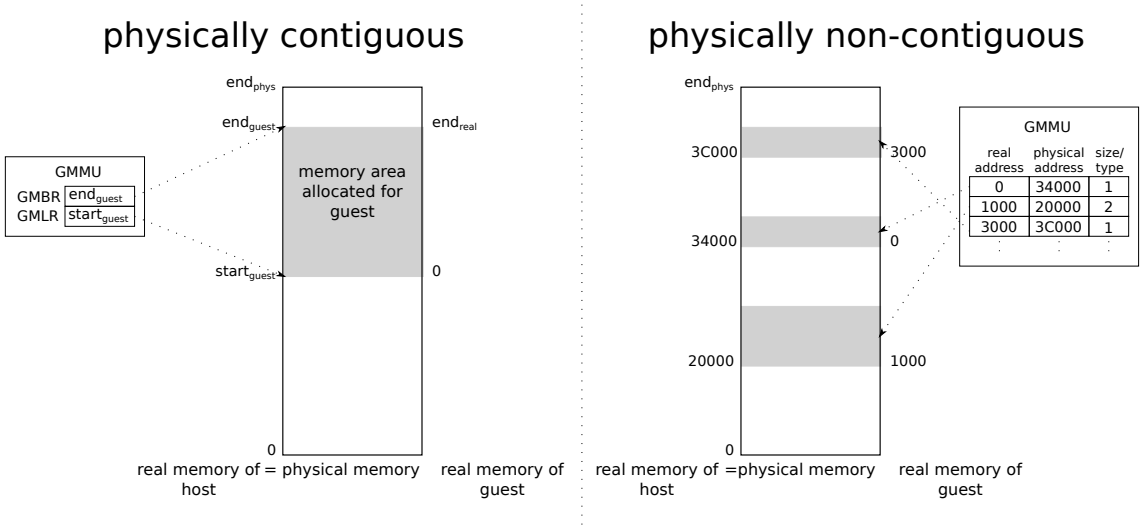


Figure 4.3.: Options to solve the problem of translating a guest's real addresses to the physically associated addresses.

contiguous chunk of memory This solution assumes the physical memory associated with the guest system to be contiguous. This assumption allows the GMMU to be very easy and straightforward. It just contains two registers, defining the first (Guest Memory Base Register, GMBR) and last address (Guest Memory Limit Register) of the physical memory area to be used by the guest system. For its simplicity its convenient to be implemented in reconfigurable logic.

page based chunks of memory (non-contiguous) The contiguous chunk of memory solution lacks some problems because of it's simplicity. There is no de-

pendency regarding the MMU of the host system. Hence, the memory area, associated with the guest system mustn't be swappable by the host system. Additionally, no differentiation regarding memory protection is possible, for example to mark some regions of the guest's memory as read only for the guest. This would be possible by introducing a GMMU, that can be seen as an extension of the host's MMU (part of the host machine's CPU). Therefore, the GMMU has to implement the same memory management mechanisms as the host MMU. For this host ISA dependency and their complexity it is more suitable to implement this type of GMMU in the static hardware part of the overall system. Nevertheless, it can also be implemented in reconfigurable logic. A GMMU based on this solution operates similar to an *Input/Output Memory Management Unit (IOMMU)* used in todays processors (see [ByMK+06] for an introduction on IOMMUs).

4.2.3. Device Issues

In this section, device virtualization related questions are discussed. As a starting point, devices are classified into two categories:

System vital, non sharable devices: These devices are essential for the functionality of the host or a guest system. Hence, sharing of such devices is usually senseless, as for example for an interrupt management controller.

Sharable devices: All other devices.

For the presented main idea, only shareable devices are of interest for further discussion. A discussion about devices is a discussion about interfaces. In the following, a device is assumed to be physically structured as given in Figure 4.4.

Figure 4.4.: General structure of a device.

A device has three interfaces:

- A **system bus interface** for connecting a device to the processor (via the system bus).
- A **controller interface** for providing a common interface and protocol for a device on a logical level, that is independent to the system, a device shall be connected to.
- A (pure) **I/O interface**, representing the "analog" wires of a device. (For an RS-232 device these are the Tx, Rx and control lines. For a hard disk, these

are the wires connected to the mechanical parts (e.g. rotors)).

A device controller's task is to transform and interpret the controller interface (and protocol) to I/O interface signals. Glue Logic's task is to transform the machine specific system bus interface and protocol to the controller specific interface and protocol.

At this point two questions arise:

1. Which interface should be shared between guest and host system?
2. How is this sharing implemented?

Interface Sharing

As a device has three different interfaces, three levels of device interface sharing are imaginable, as explained by referencing Figure 4.5:

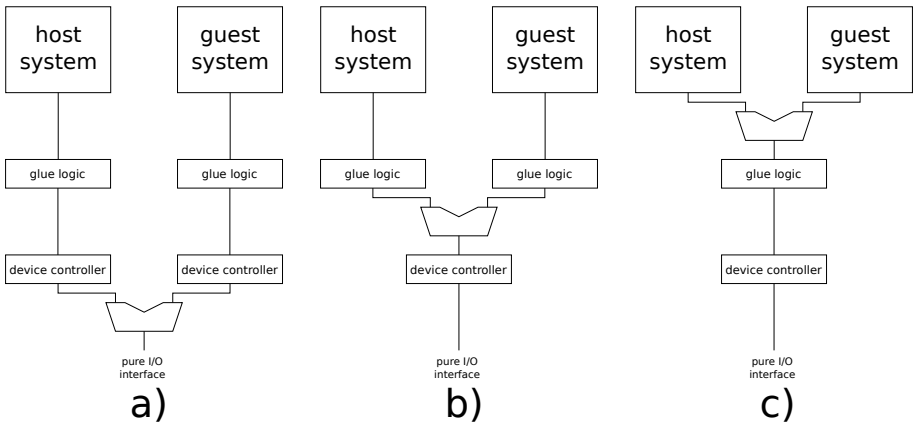


Figure 4.5.: Levels of sharing device interfaces a) shared pure I/O interface; b) shared controller interface; c) shared system bus interface.

shared pure I/O interface If the I/O interface itself is shared between host and guest machine, two device controllers need to be instantiated, resulting in unnecessary area consumption, whether or not these device controllers are instantiated in static or reconfigurable logic. This possibility is therefore only recommended for machines which use the same I/O interface but need different device controllers.

shared controller interface This solution implies the instantiation of two glue logic blocks. This is required, if guest and host machine implement different system buses.

shared system bus interface This solution is recommended for architectures,

where host and guest implement identical system buses to avoid unnecessary duplication of glue logic.

All three levels of sharing are possible. For the main idea of this thesis, the shared controller interface (Figure 4.5b)) variant is recommended due to the following reasons:

1. Assuming that enough I/O lines are available, there is no need to use the same lines in a different fashion (by using different device controllers). Additionally, taken into care the virtualization paradigm, the device provided to the guest should be the "same" as used by the host machine. "Same" is related to a device's behavior implemented by the device controller. Hence, the shared pure I/O interface solution (Figure 4.5a) is not recommended. Furthermore, most devices provide only the device controller interface. For those devices the shared pure I/O interface solution isn't applicable at all.

2. The main idea of this thesis strongly implies the possibility to instantiate a guest system bus different to the host's one. The shared system bus interface solution (Figure 4.5c) doesn't allow this, as there is only one glue logic element, providing one system bus interface.

Organizing Device Sharing

Another question concerning device sharing is, where to place the different parts of a device, in static or in reconfigurable logic?

The subsequent can be seen as a general rule:

1. All parts, that are used/needed by guest and host in the same way and therefore are physically identical, shall be implemented in static logic to reduce reconfigurable area consumption.

2. All parts, that are not used the same way, and therefore are subject of being exchanged/replaced when a switch is necessary, shall be instantiated in reconfigurable logic.

Following the recommendation of the previous section to prefer interface sharing at the device controller interface level, this implicates:

1. The device controller, is part of the static logic.

2. The guest systems glue logic shall be instantiated in reconfigurable logic. This prevents the necessity to have a guest system bus interface outside the reconfigurable area. This is important, because guest's system bus is variable as a consequence of the instantiation paradigm.

3. Hence, for each device controller, being potentially used by the guest system, a device controller interface has to be provided to the reconfigurable logic area.

4. The switching mechanism and the host glue logic, together, can be either static or preferably instantiated in reconfigurable logic. If they are instantiated in reconfigurable logic, the host system bus needs an interface into the reconfigurable area.

Implementing Interface Sharing

After having presented the different levels and organization of device interface sharing, the possibilities to implement the sharing itself are discussed.

explicit hardware switch This solution (shown in Figure 4.6) allows to physically switch between the host and guest machine. The switching itself has to be controllable by the host operating system. It can contain a mechanism, that allows the guest system to signal the need of accessing a device to the host.

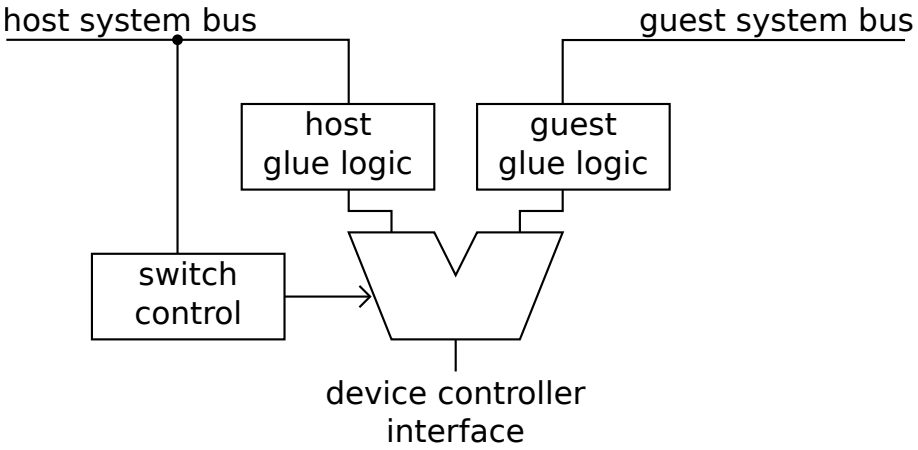


Figure 4.6.: Explicit hardware switch solution.

switching by reconfiguration Instead of physically switching between guest and host machines, the connection between a guest and a device or the host and a device can by instantiated by means of reconfiguration, as shown in Figure 4.7. The reconfiguration is initiated and controlled by the host operating system. This solution can also contain a mechanism, that allows the guest machine to signal the need of accessing a device to the host.

Figure 4.7 emphasizes another benefit of this solution. Beside reconfiguring only the connection to the guest or host system bus, the configuration stream can also include the required glue logic.

hardware supported mutual exclusion The previous solutions only provided

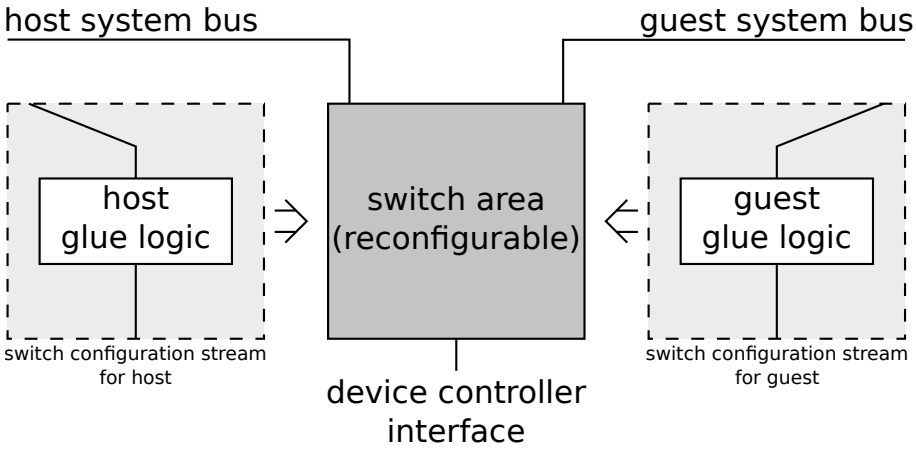


Figure 4.7.: Switching by reconfiguration solution.

mechanisms, where the guest and host machine can access a device exclusively (device is physically connected to the system bus of exactly one machine at time). The hardware supported mutual exclusion solution allows the device to be attached to the guest's and host's system bus at the same time, as shown in Figure 4.8. Mutual exclusion has to be enforced on a hardware supported

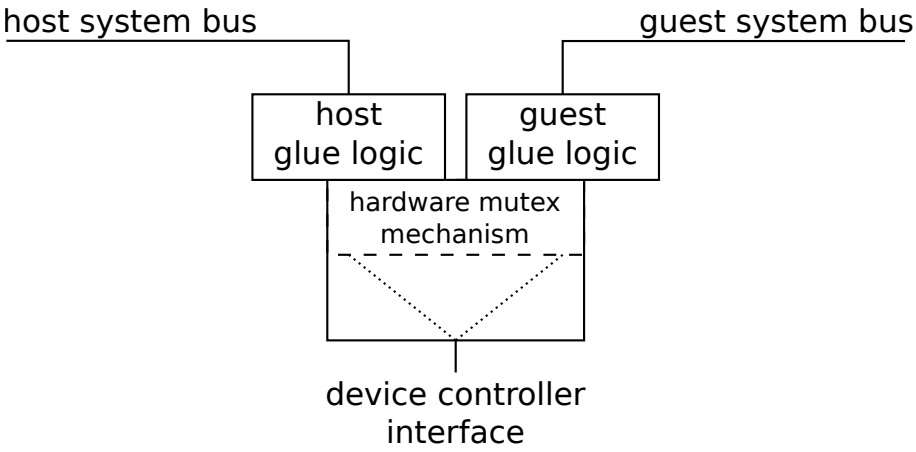


Figure 4.8.: Hardware supported mutual exclusion solution.

base, by implementing some kind of a hardware semaphore inside the switch. The host's supervising policies can be strengthened by enabling the host to suspend the mutex mechanisms.

Physically Supported Emulation

The solutions presented above try to physically share a device. For conventional system virtual machines, a physical sharing is not required, as there is only one physical system bus. On a conventional system virtualization environment, the host

operating system provides mechanisms to enable the guest system to use devices: A guest system, trying to access a device, traps to the host system, which itself supervises and handles the device access for the guest.

The mechanisms of virtualizing a physical device or emulating virtual ones, can also be applied to virtualization systems, covered by this thesis's main idea, but require adaptations.

A guest operating system cannot cause a trap to be taken on the host's processor, by means of privilege levels, because guest and host are not executed on the same processor. For this reason additional hardware has to be introduced. This additional hardware provides a physical interface to the guest system, which allows the guest to signal a device request to the host system. This interface also has to be accessible by the host system, to emulate the guest's requests. This interface looks the same, a conventional one looks like. It has to provide several addressable device register and might include interrupt line(s) to avoid busy waiting.

For the guest system, this interface behaves just like a conventional one for the interaction with a device controller. The host system (or more precisely the host operating system) needs a driver for emulating a device controllers behavior for physically non existing devices or translate the guest's request to the physical device it is associated with.

It's recommended to instantiate those physically existing virtualization interfaces in reconfigurable logic for the following reasons:

1. The guest's interface side is specific to the guest's system bus. By instantiating the (device controller) interface in reconfigurable logic, it can be easily coupled and instantiated with the necessary glue logic.

2. The number of such interfaces, required by a guest machine, is variable. A variable number can only be instantiated in reconfigurable logic, as static logic isn't changeable.

Another benefit of this solution is the possibility to not only virtualize device activities at the I/O operation level, but also at the device driver and system call level. (See page 3.2.3 for details on virtualizing I/O activity in conventional VMs.)

Virtualizing Devices

Which of the above introduces possibilities to share or provide an interface to a device is the best? There can be no final decision, because this depends on the device itself and especially on the needed time granularity of sharing. That is why some time relating definitions are required:

$t_{reconfiguration}$ - is the time needed to perform a reconfiguration process.

$T_{reconfiguration}$ depends on the size of the area to be reconfigured and the underlying technology.

t_{driver} - is the time needed by the OS, whenever a device is registered or unregistered to an operating system. This action consumes OS's processing time, as management information related to this device has to be created or disposed. The (un)registration is needed, whenever the connection between a device and a system is established or cut.

t_{usage} - is the average time a device is used by a system, before it is used by another system and therefore has to be switched to the other system.

Usually, the relation $t_{driver} << t_{reconfiguration}$ holds. Authors experience shows, that for Xilinx FPGAs, $t_{reconfiguration}$ is in the order of milliseconds and for Linux, t_{driver} is in the order of microseconds.

In the following the reasonableness of using a specific sharing functionality is discussed.

switching by reconfiguration As this solution includes to do a reconfiguration; it is only reasonable, if the relation $t_{driver} << t_{reconfiguration} << t_{usage}$ holds.

explicit hardware switch This solution requires registration and underregistration of devices within the affected operating systems. Therefore, it is only a reasonable solution, if the relation $t_{driver} << t_{usage} << t_{reconfiguration}$ holds.

hardware supported mutual exclusion If the relation $t_{usage} << t_{driver} << t_{reconfiguration}$ holds for a device, this solution is to be preferred as even device unregistration/registration would cost too much in terms of time.

physically supported emulation This sharing functionality has to be treated separately, because it shares devices logically, not physically. This logical sharing results in computational overhead for the host system, as it has to emulate device controller behavior. , *Nonexistent Physical Devices* are not sharable at a physical interface level at all. Therefore, their functionality has to be emulated by the host system, strictly requiring the *physically supported emulation* solution.

4.2.4. CPU and System related Questions

The host's hardware is given as is and the host operating system is managing this hardware resources as a conventional operating systems does. Concerning the guest, two questions, as a consequence of the virtualization paradigm, arise:

1. How is the guest hardware managed and presented as a resource in the host operating system?

2. How is the guest operating system presented in the host operating system?

Host's Representation of Guest Hardware

Regardless of what is currently configured inside a reconfigurable area, the host operating system needs the possibility to start a initiate a reconfiguration process. This is done by a reconfiguration device and an associated driver. If the reconfigurable area interfaces have to be controlled or supervised during a reconfiguration process [3], the host operating system also needs to have a device representation for controlling the reconfigurable area interfaces.

When instantiating a guest system, a lot of new interfaces might become available to the host system (memory interfaces, interfaces for device switching). The summary of those interfaces is anything the guest systems "sees" or "knows" about the guest hardware. Those interfaces need to be represented as one or more devices in the host operating system.

For engineering reasons it would be beneficial to summarize these reconfiguration specific behaviors (reconfiguration process, device representation generation and disposal after a reconfiguration) in an own OS subsystem which can be seen as virtual machine instantiation manager.

Host's Representation of Guest Operating System

In conventional virtual machine technologies, the guest operating system is represented as one (or more) processes of the host operating system (or to be more precise, the virtual machine manager).

For the host operating system of a virtualization system, based on the main idea of this thesis, the guest hardware is only a set of devices, as discussed above. For this reason a process, representing the guest's state is not suitable. Interpreting the guest operating system as a firmware, executed on those devices, would be more suitable.

Nevertheless, having a process running on the host operating system as a representation for the guest system can be beneficial. This process can acquire exclusive access to all host system related device representations of the guest system and thereby protect those devices from "misuse" by other processes running on the host operating system. Additionally, this representative process can serve as an intermediate layer between the physically supported emulation devices of the guest and the drivers of the associated physical devices.

[3] The necessary controlling mechanisms depend on the underlying technology of the reconfigurable device(s). For Xilinx related details see Xilinx Partial Reconfiguration User Guide [Xil10b].

4.3. Requirements on a Reconfigurable Logic Area

In this section the interface requirements on a reconfigurable logic area to enable the applicability of the proposed main idea are summarized. A reconfigurable area needs to have:

1. The possibility to access main memory, if guest machines memory requirements are not satisfiable by the reconfigurable area itself, e.g. by using Block RAM.
2. A device controller interface for each physical device, that can be subject of being physically used by a guest machine.
3. An interface to the host machines system bus. Several different reasons are given in the above sections.
4. At minimum one outgoing interrupt line to the host system, to enable anything configured inside the reconfigurable area (e.g a guest machine or a device sharing instance) sending interrupts to the host system.

4.4. Discussion of the Proposed Idea in Relation to Conventional System VMs

This section evaluates the proposed idea of using FPGAs to implement a new type of virtualization systems in theory. A practical evaluation of the proposed idea, based on a proof of concept demonstrator, is presented in a subsequent chapter.

No special CPU requirements for Type-1 Hypervisors

The Popek and Goldberg theorem, stating that Type 1-Hypervisors are only possible on processors, whose sensitive instructions are a subset of the privileged instructions has been discussed in section 3.2.1.

By providing a dedicated processor for executing the guest operating system, it can be executed also in privileged mode(s) of the guest processor. Therefore, the Popek and Goldberg theorem doesn't apply to reconfigurable logic based system virtualization environments.

Different ISAs - no need for emulation/ binary translation

For a virtualization system, where host and guest systems ISA are different, there is no other possibility to virtualize, as by emulation or binary translation. (See

section 3.2.1 for details.) By providing a dedicated processor for executing the guest operating system, it is possible to instantiate a processor implementing an ISA, suitable to execute the guest operating system natively.

Computational Performance

In conventional virtual machines, the virtualization enforcing strategies (guests sensitive instruction have to trap to the VMM) results in computational overhead, that reduces the possible achievable computational performance of the guest system. By providing a dedicated processor for executing the guest operating system, there is no need for this computational overhead from the processors perspective.

Nevertheless, depending on the way, memory and devices are connected and shared between host system and guest system(s), there also might be computational overhead on system VMs, based on the main idea of this thesis. E.g the physically supported emulation of devices requires host processor time to emulate a virtual device. This overhead should be smaller than the computational overhead within conventional system VMs.

The Computational Performance Problem of Soft-cores

The guest systems processor is a soft-core as a logical consequence of the proposed main idea of this thesis. As a soft-core is instantiated in reconfigurable logic, it's maximum achievable clock frequency is smaller than compared to a hard-core implementation. Hence, the achievable computational performance of a soft-core will always be smaller than a hardcore implementation. This limitation has to be taken into account for all of the above given computational performance discussions.

However, if the overall guest system computational performance isn't the dominating factor the proposed main idea is still recommended. Rather, when using virtual machine technology, computational performance shouldn't be the dominating factor at all.

Security and Supervision

Conventional virtual machines provide security by separating different guest virtual machines logically by software. Virtualization systems, following the main idea of this thesis, enforce this separation even stronger as there is physically separate hardware for the guest systems.

The introduction of the GMMU (see section 4.2.2) adds an additional hardware mechanism for limiting a guests possibility to access the hosts memory.

By using reconfigurable logic to instantiate a guest systems hardware, its possible to add special devices supervising the guests hardware. It's possible to implement those supervision devices in a way, that they are only controllable by the host system.

4.5. The Need For a Testing Framework

The presented idea of exploiting the possibilities of reconfigurable logic to build hardware based system virtual machines requires a general architecture as given in Figure 4.9 a).

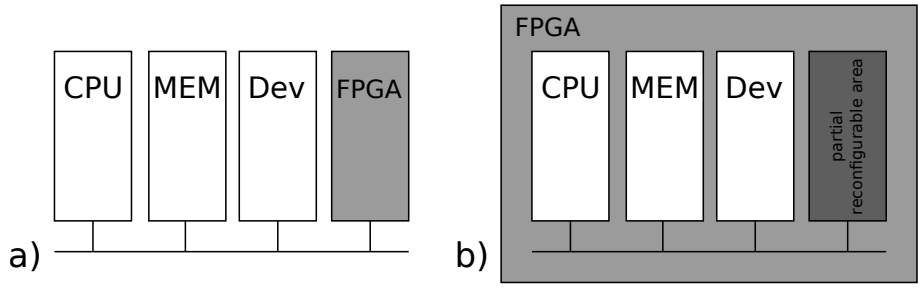


Figure 4.9.: Architectures for implementing the main idea. a) FPGA(s) as extension of a conventional system b) FPGA as prototyping environment for the overall virtualization system.

It is a conventional architecture, that is extended by reconfigurable logic. This logic fulfills the requirements to configure one or more hardware based guest system virtual machine into it (see section 4.3). The reconfigurable logic can be composed by one or more FPGAs depending on the needed reconfigurable logic area.

However, to investigate systems, based on the main idea of this thesis, it is necessary to have a testing framework, that also offers full flexibility regarding the host machines hardware. FPGAs are well suited for this purpose. The overall virtualization system hardware is configured onto a FPGA. The reconfigurable area is provided to the static part (host machine) of the overall system by means of a in-system partial and dynamic reconfigurable area, as shown in Figure 4.9 b).

Several soft-core based frameworks to instantiate an entire machine on a FPGA exist. Some of them include closed source components (especially processors, e.g. Xilinx Microblase or Altera Nios2), others are fully open source (e.g. OpenSparc). However, none of the frameworks, fulfilling all requirements to run a full fledged operating system on it, is straightforward to use or easily adaptable.

The full flexibility requirement also applies to the operating system, running on the hardware, which is another important part of a virtualization system.

For this reason, the Partial Reconfigurable Heterogeneous System (PRHS) frame-

work has been build. It consists of a hardware part (chapter 5), an adapted Linux kernel (L4PRHS, chapter 6) and software (chapter 7) to extent the kernel to a full fledged operating system, that can be used as guest and/or host operating system.

5. PRHS Framework - Hardware

In this section, the hardware components for the Partial Reconfigurable Heterogeneous System (PRHS) are presented. A general overview is given in section 5.1. The PRHS Bus, connecting all devices of PRHS framework, is defined in section 5.2. The PRHS Processor - ARM Instruction Set (PRHSp-A) as central processing element of the PRHS framework hardware is presented in section 5.3.

The main memory subsystem devices of the PRHS framework hardware is given in section 5.5. The different devices, available in PRHS framework, are shown as well. They are differentiated as *base system devices* (section 5.4) and *platform specific devices* (section 5.6). The *partial reconfiguration extension* as partial reconfiguration enabler for the PRHS framework is given in section 5.7. Finally, composed hardware modules that allow easy reuse are presented in section 5.8

5.1. General Overview on PRHS Hardware

The tasks of the PRHS hardware is to provide:

- All elements to build an entire system on chip in an easy and fast fashion.
- Different platforms should be supported. A FPGA and devices, combined on a board are seen as a platform.

To achieve those goals PRHS framework has been designed with three fundamental system levels:

1. small system
   The small system combines the PRHSp-A with Block RAM as main memory and an UART for user interaction. It might serve as a starting point for small embedded systems. It fits entirely onto a FPGA, enough area provided. It should be the first step, to get this system level running on a new platform, PRHS framework shall be ported to. Focus should lie on system clock management in such a case.

2. base system
   Extends the small system by a platform independent main memory subsystems. This subsystem allows to include a platform specific memory controller

and the associated memory into PRHS hardware. The base systems contains all necessary devices to run a customized Linux (L4PRHS) on it.

3. reconfiguration system
   Extends the base system with a partial reconfiguration extension. This enables to use in-system partial and dynamic reconfiguration within PRHS hardware.

The particular components of the systems are explained in detail in the following sections.

5.2. PRHS Bus Definition

The *PRHS Bus* interface is inferred by the requirements for the entire system itself. These requirements are:

1. The system is a 32 bit system. Therefore, the data lines and the address lines of the bus are 32 bit wide.

2. The *PRHS Bus* shall support pipelined processors. To entirely utilize the performance benefits of pipelined processors, it is necessary to get an answer for a memory request in the same clock cycle as the request occurs [HP07][PH09]. Therefore, a dedicated data line from processor to device (write data) and from device to processor (read data) is implemented.

3. Three transfer modes shall be possible: read, write and swap. A swap is a transaction that performs a read and a successive write, that mustn't be interruptible. It is a necessary hardware element to support mutexes and semaphores in an operating system [Tan07][HP07][PH09].

4. A transfer width of 32 bits (word), 16 bits (halfword) or 8 bits (byte) shall be supported.

5. The answer lines (output signals) of the different devices shall be connectible as pull-up bus lines.

6. All *PRHS Bus* participants work on the same clock.

5.2.1. PRHS Bus Interface

The above requirements result in the following *PRHS Bus* interface:

```
-- signals from master (processor) to slaves (device/memory)
signal sePRHSrequest    : std_logic;
signal scPRHSWidth      : prhsBusWidth;
signal scPRHSoperation  : prhsBusOperation;
signal scPRHSaddress    : ADDRESS;
signal sdPRHSdataMaster : DATA;

-- signals from slaves (devices/memories) to master (processor)
signal sdPRHSdataSlave  : DATA;
signal senPRHSdone      : std_logic;
```

Listing 5.1: Signal declaration for PRHS bus.

using the self-defined data types:

```
type prhsBusOperation is (OPread, OPwrite, OPswap);
type prhsBusWidth is (Width_Word, Width_HalfWord, Width_Byte);

subtype DATA        is std_logic_vector(31 downto 0);
subtype ADDRESS     is std_logic_vector(31 downto 0);
```

5.2.2. PRHS Bus Protocol

The following table explains the meanings of the above given signals in detail.

Name	Driver	Description
sePRHSrequest	master	enable signal asserted by master to initiate a transfer
scPRHSWidth	master	control signal for data width: `Width_Word` for all 32 bits of data lines, `Width_HalfWord` for lower 16 bits of data, `Width_Byte` for lower 8 bits of data
scPRHSoperation	master	control signal for transaction type: `OPread` for data transfer from device to master, `OPwrite` for data transfer from master to device, `OPswap` for data exchange between master and device
sdPRHSaddress	master	control signal for transfer to address the device registers/ memory cells on a per byte basis; PRHS framework only supports little endian data ordering
sdPRHSdataMaster	master	data signal: the data transfered from master to device, value is irrelevant for read transfers; unused data bits (according to `scPRHSWidth`) shall be set to '0'

sdPRHSdataSlave	device	data signal: the data transfered from device to master, value is irrelevant on write transfers; signals of different devices can be connected on a shared pull-up line; unused data bits (according to `scPRHSWidth`) shall be set to '0'
senPRHSdone	device	active-low enable signal of a device to signal transfer completion to master; signals of different devices can be connected on a shared pull-up line

Read Transfers

When initiating a read request, the *PRHS Bus* master sets `scPRHSoperation` to `OPread`, `scPRHSWidth` and `scPRHSaddress` to the appropriate values. The value of `sdPRHSdataMaster` doesn't matter for a read transfer. The request itself is initiated by the master when the `sePRHSrequest` signal is set to '1'.

It is now up to the device, responsible for the address given with `scPRHSaddress`, to set `sdPRHSdataSlave` to the right value and signal the end of transfer by setting `senPRHSdone` to '0'. On the next rising edge of the bus clock, the transfer is finished for master and device and the next transfer can be initiated by the master instantaneously. If `senPRHSdone` is asserted to '0' by the device in the same clock cycle as `sePRHSrequest` is asserted to '1' by the master, the transfer is called to be an immediate transfer. If the device doesn't immediately signal the end of the transfer by asserting `senPRHSdone` to '0' in the same cycle the master sets `sePRHSrequest` to '1' the transfer is called non-immediate.

The master can change the values of `scPRHSoperation`, `scPRHSWidth`, `scPRHSaddress`, `sdPRHSdataMaster` and also `sePRHSrequest` on a successive rising edges of the bus clock. Nevertheless, as long as `sePRHSrequest` remains asserted, the other signals will not change. This behavior mustn't affect the already initiated transfer of a device. Figure 5.1 and 5.2 give same examples for valid read transfers.

Devices not responsible for the requested address should not drive the signals `sdPRHSdataSlave` and `senPRHSdone`.

Write Transfers

Write transfers turn around the direction data is transfered. Therefore, they are almost the same like read transfers, except that `sdPRHSdataMaster` is now an important signal. It contains the data to be transfered from master to device. At the same time, the value of `sdPRHSdataSlave` is meaningless if `senPRHSdone` is asserted to '0' by the device, responsible for answering the request. Figure 5.3 gives examples for write transfers.

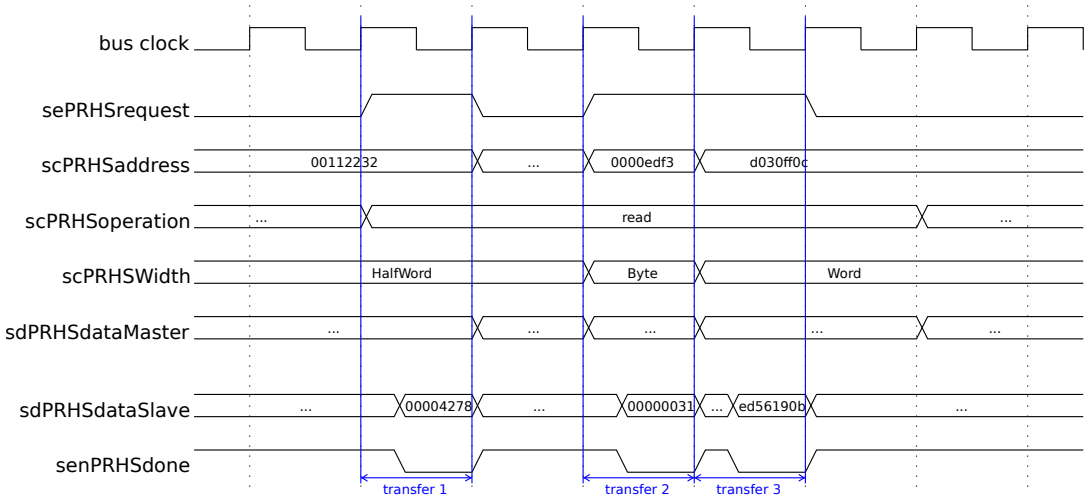


Figure 5.1.: PRHS Bus read transfers with immediate answer for 3 transfers.

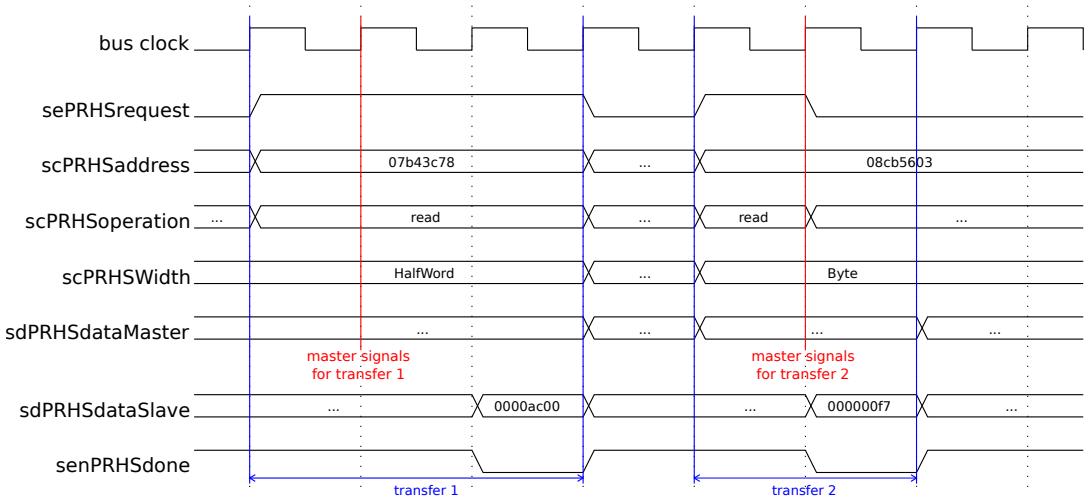


Figure 5.2.: PRHS Bus read transfer with non-immediate answer for 2 transfers and changing master signals during transfer.

Swap Transfers

Swap transfers combine a read and write transfer to a given address that is not interruptible. Therefore, the values of `sdPRHSdataMaster` and `sdPRHSdataSlave` are relevant both. `sdPRHSdataSlave` contains the old value of the device register or memory cell addressed by `scPRHSaddress` before the value of `sdPRHSdataMaster` had been written into it. Figure 5.4 gives examples for 2 consecutive swap transfers on the same device register/memory cell.
The same address and width is used among all transfers. Transfer 1 writes a value to the addressed device register/memory cell. Second transfer is a non-immediate swap, writing a new value to the same register/cell and getting back the value written during the first transfer. The returned value of transfer 3 is the value written by

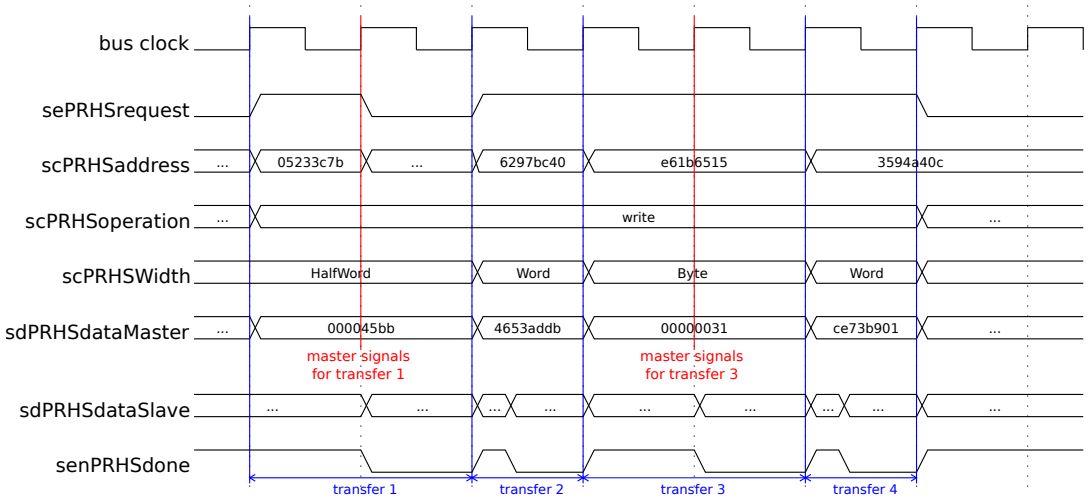


Figure 5.3.: PRHS Bus write transfer examples.

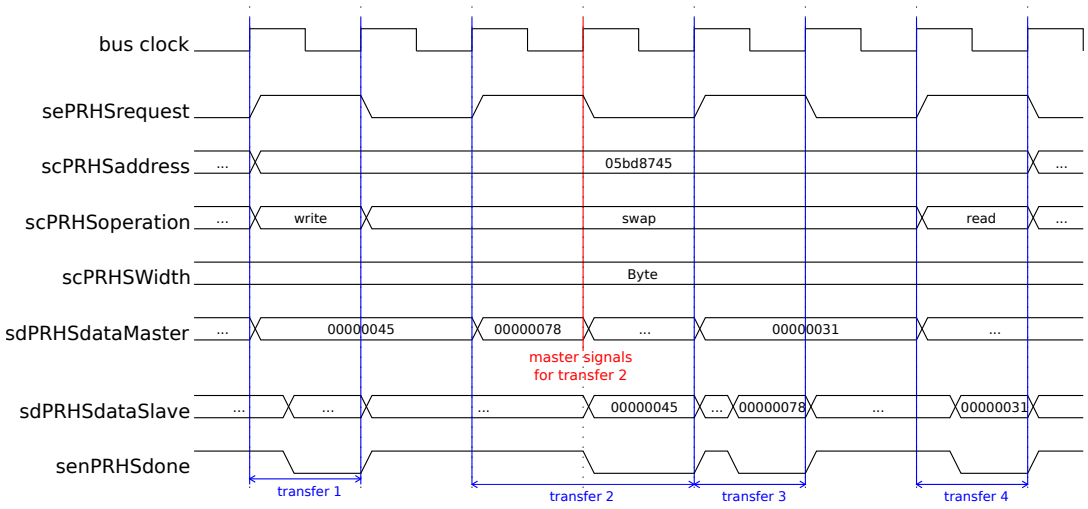


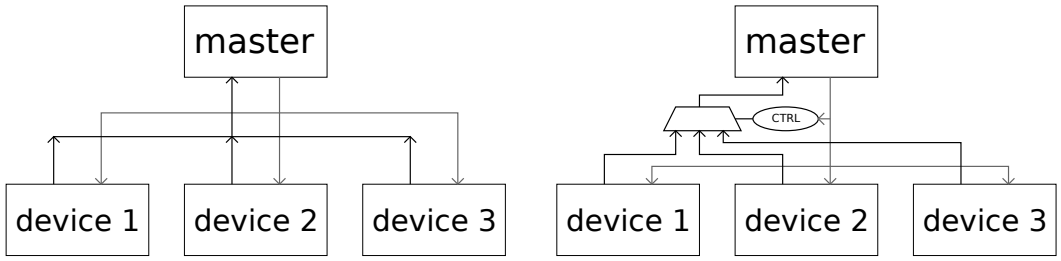
Figure 5.4.: PRHS Bus swap transfer example.

transfer 2. The final read now gets the value written during transfer 3.

Primary and Secondary PRHS Bus

According to [HP07](page 390) and [Fur02](page 278) devices can be connected to a master in several ways. Figure 5.5 shows the two possible extrema.

The first connection solution is a shared Bus (Figure 5.5 (a)). The main advantage of this solution is the minimal hardware overhead for the devices. Their outputs should be deactivatable to ensure that only one of the devices is driving the shared bus lines on request. The main disadvantage of this solution is the shrinking maximum achievable transfer rate (bus clock frequency), when more and more devices are



(a) Shared Bus (b) Multiplexed Connection

Figure 5.5.: Schemes for connecting devices with a master.

connected to the shared bus lines.

The second connection solution is to multiplex the outputs of the several devices resulting in point-to-point connections (Figure 5.5(b)). This solutions doesn't lack the problem of from shrinking transfer rates, when more and more devices are added. However, as more and more devices are connected, the multiplexing logic and therefore area consumption increases.

For the PRHS framework a mix of the above presented solutions has been chosen on the basis of the following statements:

- A systems processor is accessing the main memory most of the time. In comparison, device accesses are seldom.
- Hence, main memory access should be fast and therefore has to exploit the possibility for immediate answers on *PRHS Bus*. This requires a low latency connection, because a transfer might be accomplished at the same clock cycle as it is requested (immediate transfer). For this reason, main memory gets is own PRHS Bus (primary PRHS Bus).
- On the other hand, seldom devices accesses can be slow without affecting the overall performance at all. Therefore, input/output devices are connected via a shared bus (secondary PRHS Bus) only supporting non-immediate transfers.
- Primary and secondary PRHS Bus are connected to the processor using a multiplexed scheme as explained above, resulting in the architecture as given in Figure 5.6.

A detailed description of the PRHS BusCtrl is given in section 5.4.2

Transfer Aborts

As already mentioned, all device registers are memory mapped into the 32 bit physical address space of the processor.

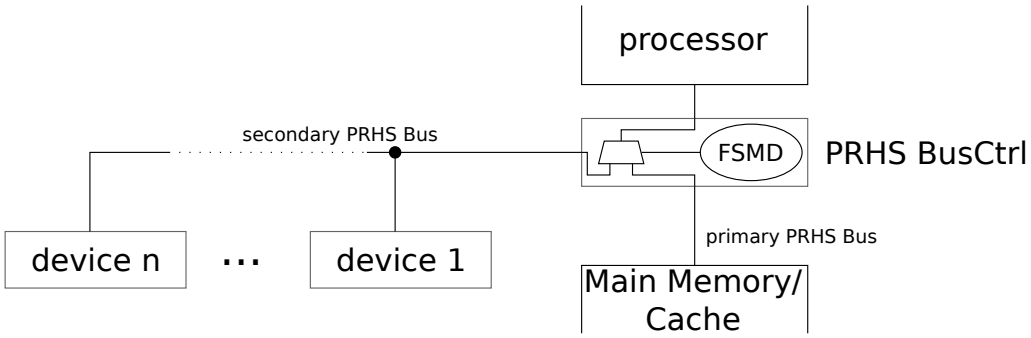


Figure 5.6.: Primary and secondary PRHS Bus.

It might occur a PRHS Bus request to an address (using `scPRHSaddress`) that is not used by the memory or any device. To prevent the processor from infinite blocking, the PRHS BusCtrl will signal the processor a transfer abort, if no answer is given (via asserting `senPRHSdone`) on a request after a predefined number of PRHS Bus clock cycles.

5.3. PRHSpA - PRHS Processor ARM ISA

The PRHSp-A is the central processing element for PRHS framework. Its general architecture is given in Figure 5.7.

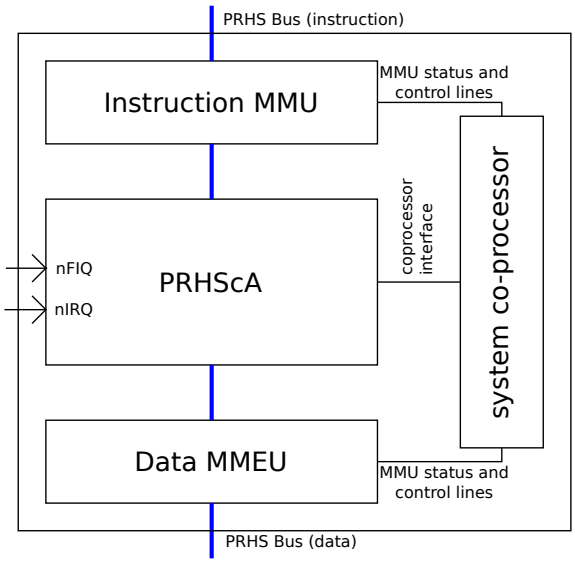


Figure 5.7.: General architecture of PRHSpA.

It is based on the information given in [Adv96] to implement a processor, that is instruction set compatible with the ARM8 ISA and is usable in the PRHS framework.

To make adaptations of an operating system (see chapter 6) more easier, selected key functionality to support memory management mechanisms, as described in [Adv96], is implemented.

The different components are explained in more detail in the following sections.

In opposition to the original processor as presented in [Adv96], PRHSp-A doesn't implement an on-processor cache, a write buffer or a prefetch unit.

5.3.1. PRHScA - PRHS Core ARM ISA

The PRHS Core - ARM Instruction Set (PRHSc-A) is the central processing element inside the PRHSp-A.[1] It implements the ARM8 ISA as presented in detail in [Adv96] and [Fur02]. It provides a dedicated instruction and data memory interface (based on the PRHS Bus definition of section 5.2). In addition it provides a co-processor interface and two interrupt lines (nFIQ - fast interrupt, higher priority than nIRQ - normal interrupt).

The PRHSc-A is implemented as a 3 stage pipeline processor core, as given in Figure 5.8 (page 56).

A major challenge in designing a pipelined processor are pipeline hazards. For PRHSc-A the different types of pipeline hazards and their solutions are discussed in the following.

Structural hazards occur, when the hardware cannot support the combinations of instructions that should be executed in one clock cycle. ARM ISA contains multi cycle instructions[2] by design. For example the *block transfer* instructions. These instruction type tries to copy a list of registers to or from main memory via the data memory interface. However, the data memory interface is not able to transfer more than one, on newer ARM architectures two, register to or from main memory concurrently. Hence, a *block transfer* is a multi cycle instruction.

ARM ISA includes complex addressing schemes, as for example:

```
LDR R0,[R1,R2,LSL#4]; load contents of R1 + R2 * 8 to R0
```

Implementing this instruction as a one cycle instruction would result in a very long data path, as an immediate value is used to shift a register, add the result

[1] Acknowledgement: The current PRHSc-A is a redesigned implementation of a processor core, that was originally implemented in cooperation of Klaus Hildebrandt and the author of this thesis. The old version was an unpipelined processor implementation of the ARM7 instruction set. Several parts of this old version have been re-used and re-engineered by the author of this thesis for performance reasons. The current PRHSc-A was implemented by the author alone.

[2] Multi cycle instructions take more than one clock cycle per pipeline stage to be executed.

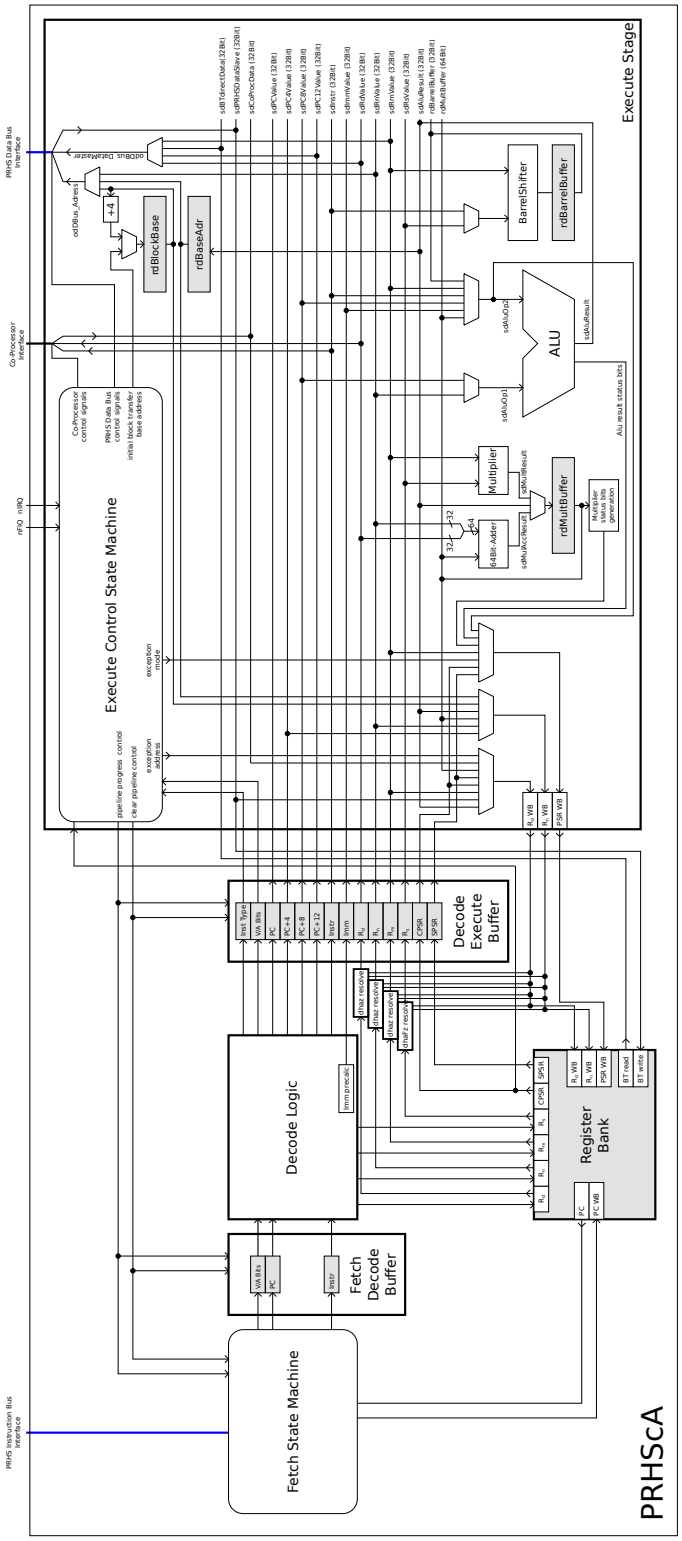


Figure 5.8.: General architecture of the three stage pipelined PRHScA. Gray boxes represent registers. Other elements are combinational logic or state machines.

to the value of another register and use this sum as address for accessing memory. A long data path results in a lower maximum achievable clock frequency. Therefore, selected instructions have been implemented as multi cycle instructions in PRHSc-A, whereas the original ARM 810 provided them as single cycle instructions. At this point, PRHSc-A offers room for improvement by using more than 3 pipeline stages. However, overall performance is out of the scope of this thesis.

Data hazards arise from from data dependencies between subsequent instructions. For a three stage pipeline only two subsequent instructions can be affected. For example, one of the two possible resulting registers of an instruction is used as one of the four possible source registers of the next operation. These data hazards are resolved by *forwarding* the resulting register values of an instruction. This is implemented by the *data hazard resolver* (dhaz resolve) modules.

Control hazards arise every time, the program counter is modified by an instruction. On ARM architectures, there are several instructions that affect the program counter (e.g. branches or data processing instructions targeting the program counter). To make matters worse, each ARM instruction can be conditional by the design of the ARM ISA. In consequence, a prediction unit is not straightforward to implement. For the current PRHSc-A implementation, a modification of the program counter results in clearing the entire pipeline. A further improvement of PRHSc-A clock cycles per instruction (CPI) rate is possible here, but out of the scope of this thesis.

In the remainder of this section, the different parts of PRHSc-A are briefly described.

Fetch State Machine

The task of the fetch stage machine is to retrieve an instruction form memory, based on the current program counter. The fetch state machine is given as state chart in Figure 5.9. For better understanding, it is important to consider the following:

- The pipeline control signal (*clear pipeline* and *pipeline progress*) are mutual exclusive.
- The done signal for a PRHS instruction bus request can be given immediate (in the same clock cycle the request was issued) or non-immediate (number of clock cycles is not predictable by PRHSc-A).
- The abort signal for a PRHS instruction bus request is non-immediate.
- The done signal for a successful PRHS instruction bus request and the abort signal on a non-successful PRHS instruction bus request are mutual exclusive.

- To keep the PRHS instruction bus in a clean state, each PRHS instruction bus request has to be finished by a done or abort signal. Hence, a pipeline control command (*clear pipeline* and *pipeline progress*) is ignored (not acknowledged) by the fetch state machine until a done or abort signal is occurring for a pending PRHS instruction bus request.

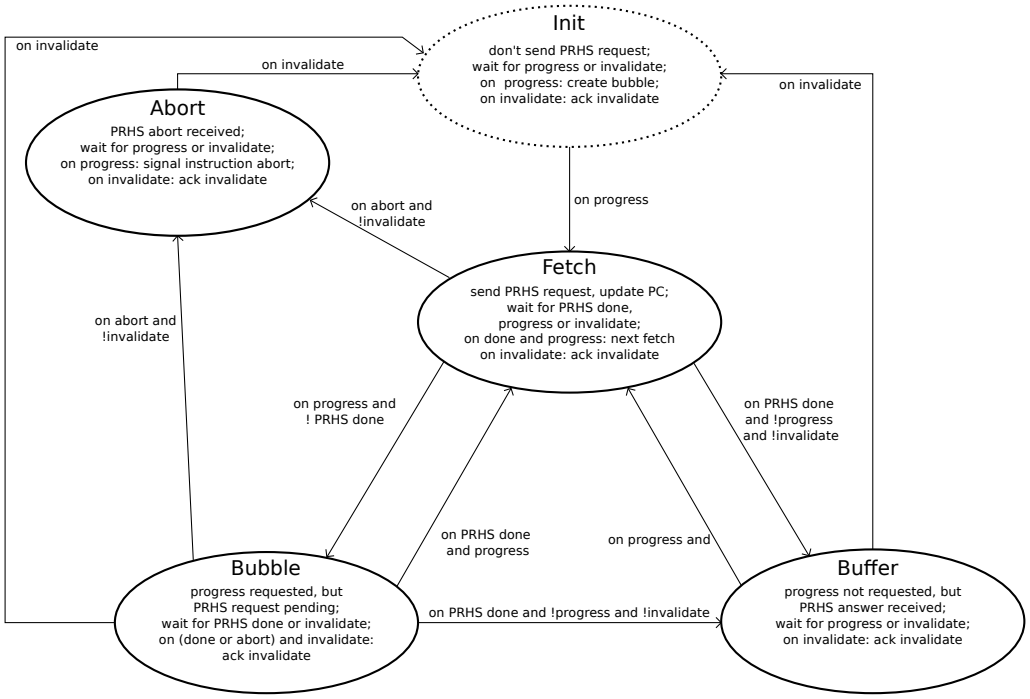


Figure 5.9.: State chart for the fetch state machine.

Fetch Decode Buffer

The task of the fetch decode buffer is to hold the next instruction to be decoded. In addition it holds the corresponding program counter of the instruction. The contents can be marked as a bubble by setting the valid bit (V) to '0'. If instruction memory signaled a data abort during the fetch stage (fetch state machine), the abort bit (A) is set.

A *clear pipeline* request sets the valid bit to '0'. Thus creating a bubble. On a *pipeline progress* command, the incoming values, generated by the fetch state machine, are registered.

Decode Logic

The task of the decode logic is to decode the instruction to be executed next. This includes the decoding of the instruction type, the calculation of the program counter offset values (+4, +8 and +12) and the precalculation of immediate values, encoded in the instruction.

In addition, the select values for the register bank are decoded and signaled to the register bank to retrieve the register values needed by the instruction. This includes up to four general purpose register (R_d, R_n, R_m and R_s).

Register Bank

The task of the register bank is to hold all 31 general purpose registers and six status registers. For further details on the arm8 specific register set organization and the processor mode dependencies see [Adv96] or [Fur02].

The register bank offers four register read ports (R_d, R_n, R_m and R_s) for the decode phase to retrieve the required register values and send out the $CPSR$ (current program status register) and $SPSR$ (special program status register).[3]

The register bank also offers write back ports for two general purpose registers (R_dWB, R_nWB) and the PSR (program status register). The PSR write back port allows to select whether CPSR or SPSR is the write back target.

For block transfer (BT) instructions the register bank provides a BT read and write port to allow the execute stage to read and write the required registers directly.

Finally, the register bank also provides a dedicated PC (program counter) read and write back port to be used by the fetch State machine.

Decode Execute Buffer

The task of the fetch decode buffer is to hold the next instruction to be executed. In addition it holds the corresponding program counter of the instruction and offsets to the program counter (+4, +8 and +12) for usage during the execute stage. The required input registers and immediate values, used by the instruction are also buffered in the Decode Execute Buffer for the corresponding instruction. The contents can be marked as a bubble by setting the valid bit (V) to '0'. If instruction memory signaled a data abort during the fetch stage (fetch state machine), the abort bit (A) is set.

[3] In *user* and *system* mode, SPSR and CPSR are the same for PRHSc-A implementation.

A *clear pipeline* request sets the valid bit to '0'. Thus creating a bubble. On a *pipeline progress* command, the incoming values, generated by the decode logic and the register bank, are registered.

Execute Stage

The data path of the execute stage is presented in detail in Figure 5.8 (page 56). The control signals for the execute stage and the *clear pipeline* and *pipeline progress* control lines are generated by the *Execute Control State Machine*, which is given as state chart in Figure 5.10.

5.3.2. System Co-processor

ARM ISA supports up to 16 co-processors, attached to a processor core and therefore defines special co-processor commands and an appropriate co-processor interface. Co-processor 15 is recognized as a special one, the system co-processor. ARM ISA based processors usually include a system co-processor, to control processor functionality of dedicated devices. The registers interpretations for these system co-processors vary from processor to processor for this reason.

The PRHSp-A system co-processor register interpretation is based in the system co-processor described in [Adv96] (chapter 5), but only implements functionality needed by PRHSp-A.

The following table gives a summary on the register interpretation of the system co-processor of PRHSp-A.

Register	Read	Write
0	Processor ID	Undefined
1	Control	Control
2	Translation Table Base	Translation Table Base
3	Domain Access Control	Domain Access Control
4	Undefined	Undefined
5	(Data) Fault Status	Ignored
6	(Data) Fault Address	Ignored
7[4]	Ignored	Ignored
8	Undefined	TLB Control (flush TLBs)
9 - 15	Undefined	Undefined

Each entry marked as *Undefined* will cause a data abort. Each entry marked as *Ignored* will finish successfully, but has no effect (write)/results in zero (read).

[4]Cache Control on original processor[Adv96]

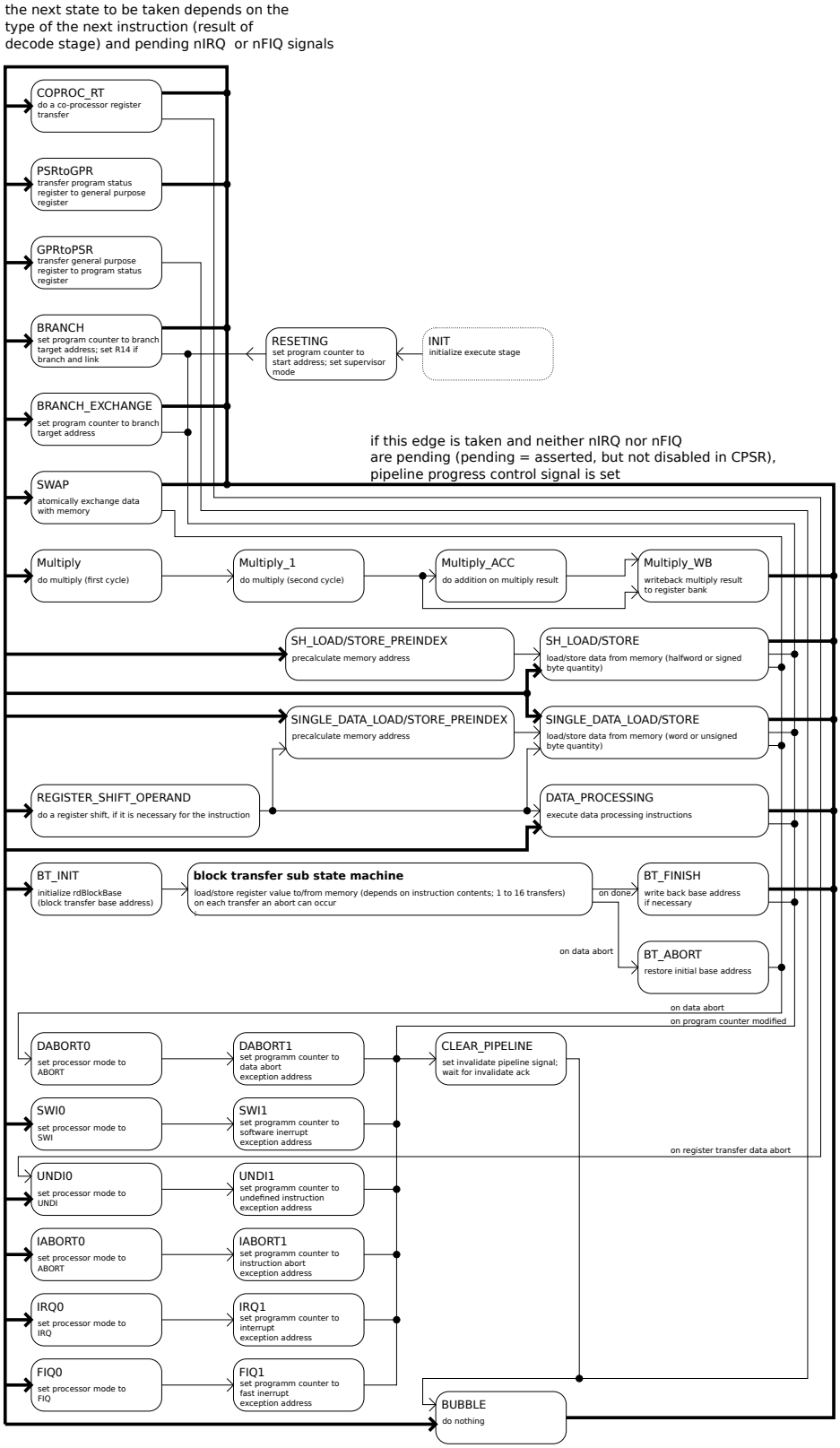


Figure 5.10.: State chart for the execute control state machine.

A read access to the ID register (register 0) is answered with the processor ID ($0xff008100$).

The following table shows the bit interpretation of the control register (register 1):

Bit	Description
31:10	Ignored
9	ROM-protection-Bit: MMU related
8	System-protection-Bit: MMU related
7:1	Ignored
0	MMU-enable-Bit: '0' MMU disabled, '1' MMU enabled

5.3.3. MMU - Memory Management Unit

The PRHSp-A contains two Memory Management Entities (MMU), one for the data interface and one for the instruction interface of the processor. Their primary task is to translate virtual addresses into physical ones and enforce a protection strategy related to this virtual address space. ARM based MMUs are *architected page table MMUs*. This means, the operating system is responsible for placing page tables into main memory. The MMU inspects those page tables on a TLB (Translation Look-Aside Buffer) miss to fetch an appropriate TLB entry. For a detailed introduction to MMUs see [HP07] or [SN05b].

The PRHSp-A MMUs implement the architected page table functionalities given in [Adv96] (chapter 8), except the following:

Cache and Write Buffer behavior As PRHSp-A doesn't have a cache or write buffer, no functionality related to those devices is implemented in PRHSp-A MMUs.

Fault generation PRHSp-A MMUs don't generate vector, alignment or terminal faults.

Taken the above statements as requirements results in the MMU state machine as given in Figure 5.11.

5.4. Base System Devices

The PRHSpA processor presented in the previous section isn't usable standalone to run an operating system or even a small program on it. A minimum of additional devices, summarized as base system devices, are required for this purpose. Those devices are presented in the subsequent sections. They are independent from the platform, a PRHS based system is running on.

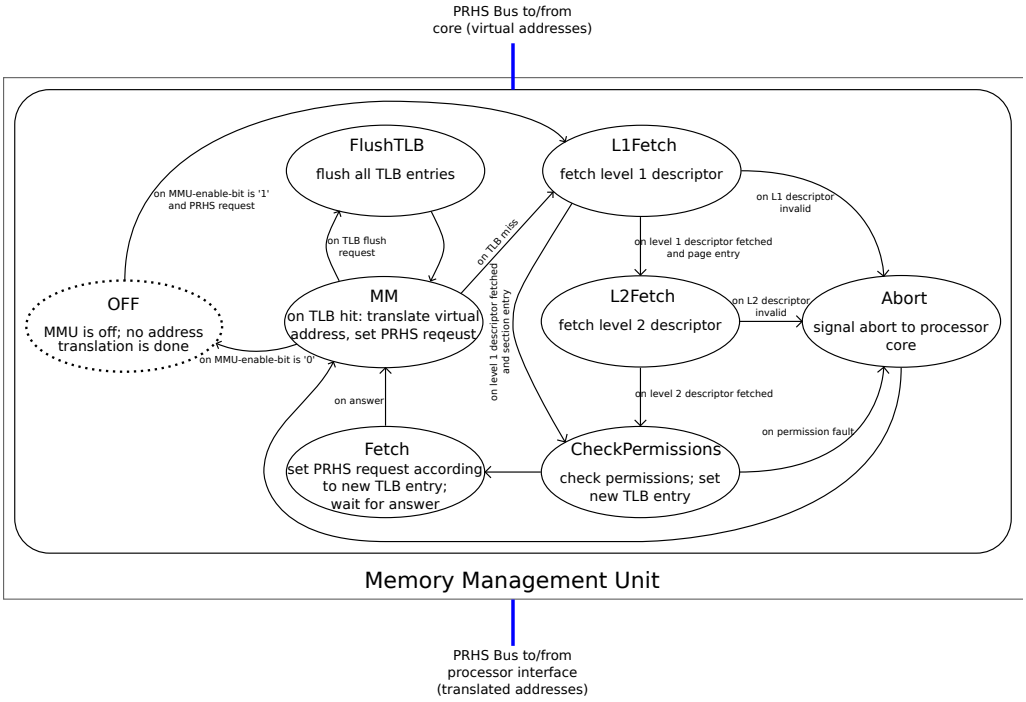


Figure 5.11.: State machine for the memory management unit.

5.4.1. I/O Devices Fundamentals

A lot of I/O devices are included in the PRHS framework. For the following reasons an I/O device cannot be directly connected to the PRHS Bus:

1. PRHS Bus clock and I/O device clock may differ, resulting in necessary clock domain crossing problems.

2. PRHS Bus can only address a limited number of up to 32 bits simultaneously. I/O device inputs and outputs might implement another data width.

3. I/O operations might take a while till they are finished. The PRHS system itself should not generally block further execution in those cases.

For this reasons, a general mechanism is implemented to connect I/O devices to PRHS framework hardware as presented in Figure 5.12.

The *clock domain management* block is optional and responsible for solving clock domain crossing issues. The *register state machine* provides memory mapped registers on the PRHS Bus. The contents of those registers are "wired" to the control and data signals of the I/O device itself.

Functionality, interfaces, protocols and clock domain management for I/O devices, included in the PRHS framework, are given on a per device basis in the following

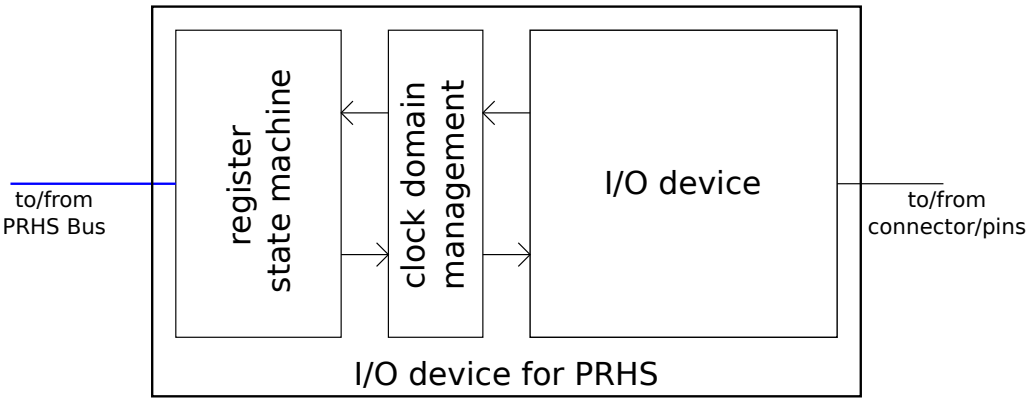


Figure 5.12.: General structure of connecting I/O devices to PRHS Bus.

sections. The functionality of the register state machine is common and fundamental to all I/O devices and therefore defined beforehand. In the I/O device specific section, only the bit and byte meanings of the register state machine registers are given.

Register State Machine Interface

A template for implementing the PRHS Bus interface of I/O register state machines is included in the hardware part of PRHS framework as given in listing 5.2.

```
entity templateStateMachine is
  generic (
    GEN_BaseAddress : ADDRESS := X"c8000000";
    GEN_AddressMask : ADDRESS := X"fffffff0"
  );
  port (
    iSysClk          : in  std_logic;
    iSysClkEn        : in  std_logic;
    iReset           : in  std_logic;

    -- PRHS Bus master signals
    iePRHSrequest    : in  std_logic;
    icPRHSWidth      : in  prhsBusWidth;
    icPRHSoperation  : in  prhsBusOperation;
    icPRHSaddress    : in  ADDRESS;
    idPRHSdataMaster : in  DATA;
    -- PRHS Bus device signals
    odPRHSdataSlave  : out DATA;
    oenPRHSdone      : out std_logic

    -- add I/O device control signals below this
  );
end templateStateMachine;
```

Listing 5.2: Entity declaration for register state machine.

As the I/O devices all reside on the secondary PRHS Bus, none of them is allowed to support immediate PRHS Bus transfers (refer to page 52 for details). For this reason the register state machine is implemented as a two state finite state machine as shown in Figure 5.13.

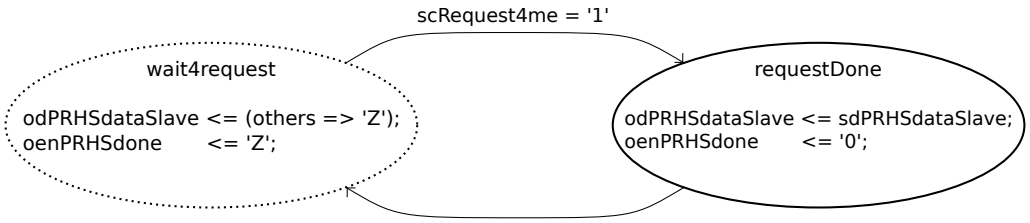


Figure 5.13.: State chart for I/O register state machine.

The signals `scRequest4me` and `sdPRHSdataSlave` are internal signals, where

```
scRequest4me <= '1' when iePRHSrequest = '1' and
                ((icPRHSaddress and GEN_AddressMask) = GEN_BaseAddress)
3               else '0';
```

and `sdPRHSdataSlave` is set to the appropriate value, when the state machine switches from *wait4request* to *requestDone*. "Appropriate" refers to `icPRHSWidth`, `icPRHSoperation` and the masked (lower) bits of `icPRHSaddress`.

The generic `GEN_BaseAddress` contains the memory mapped start address of the I/O device. The address range used for the device can be calculated using `GEN_AddressMask`. Defining those values as generics and not as constants allows multiple instantiations of a device with different memory mappings.

5.4.2. PRHS Bus Controller - Multiplexing primary and secondary PRHS Bus

According to the information, given for the *PRHS Bus* in section 5.2.2 on page 52, the *PRHS Bus Controller* device has two main functions:

1. Multiplex PRHS Bus transfers of a processor onto the corresponding primary or secondary PRHS Bus.
2. Provide a timeout mechanism to prevent the system from trapping into infinite waiting on transfer requests to unmapped memory/device register addresses.

The first is achieved by implementing a bus multiplexer state machine. The second requirement is fulfilled by implementing a bus timeout state machine.

The general structure, including both state machines, is given in Figure 5.14.

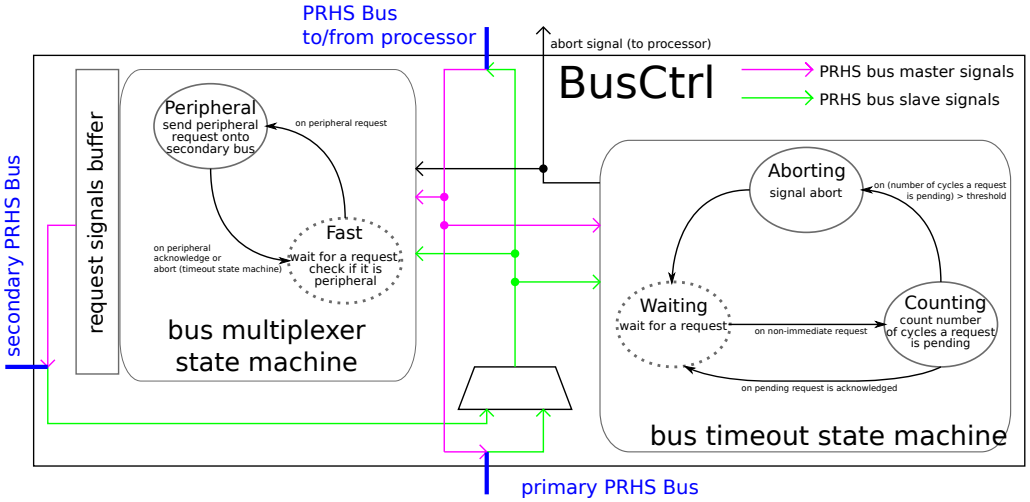


Figure 5.14.: General structure of the PRHS Bus Controller.

As can be seen in Figure 5.14, the PRHS Bus master signals are buffered in the *request signals buffer* if a PRHS Bus transfer targets a peripheral device. A PRHS Bus transfer is a peripheral request, if the requested address is greater or equal than `GEN_secondaryBusStart`. `GEN_secondaryBusStart` is a generic to easily change the split of address space into primary and secondary bus at synthesis time, where all addresses less than `GEN_secondaryBusStart` reside on the primary bus and all addresses greater or equal than `GEN_secondaryBusStart` on the secondary bus.

As a side effect of the request signal buffering, transfers requested on the primary bus are not visible on the secondary bus. On the contrary, secondary bus transfers are visible on the primary bus. This design decision results from the demand, that the primary PRHS Bus shall support immediate transfers (refer to page 52 for details). Adding any mechanism to make secondary bus transfers invisible to primary PRHS bus would increase latency between request and answer of an immediate transfer on primary bus and results in the reduction of the maximum achievable system frequency.

The bus timeout state machine is independent of the bus multiplexer state machine. It counts the number of clock cycles between the occurrence of a `iePRHSrequest` signal and the corresponding `oenPRHSdone`. If the counter reaches a value given with the generic `GEN_TimeoutValue` it signals a transfer abort to the processor.

The *PRHS Bus Controller* device doesn't offer any I/O mapped registers accessible by the processor. Therefore, it isn't "seen" by the operating system, except the occurring aborts it signals.

5.4.3. BusComponentStatus - Managing Device Discovering

BusComponentStatus device has two primary functions:
It provides a mechanism to enable software to identify,

1. on what kind of board or FPGA the system is running.
2. which devices are connected to PRHS Bus.

To make the first possible, *BusComponentStatus* offers the possibility to query for the board/FPGA, the system is running on. The information is "hardwired"' into *BusComponentStatus* at synthesis time using a generic.

As PRHS Bus (refer to definition given on page 52) doesn't support some kind of auto-discovering or hot-plugging functionality, another way to identify, which devices are attached to PRHS Bus is implemented. The information is encoded on a one bit per device scheme. It is currently limited to 32 bits (the width of the data signals of PRHS Bus).

For the given reasons, *BusComponentStatus* is neither connected to an external nor to an internal "real" device. Therefore, it only implements a PRHS Bus interface and a register state machine as described on page 64.

register interpretation

Each access to the *BusComponentStatus* registers is interpreted as a read word. Therefore, `icPRHSWidth` and `icPRHSoperation` is ignored and `idPRHSdataMaster` is meaningless.

system type Register (`GEN_BaseAddress`)

bit	R/W	description
[31:0]	R	Contains integer value for the corresponding boardFPGA.

attached devices register (`GEN_BaseAddress` + 0x4)

bit	R/W	description
[31:0]	R	Device present bit. See Appendix B for devices, board/FPGA and bit association.

Access to any other register address than the above given, results in an unpredictable answer of the device.

Usage of this device by Linux for PRHS (L4PRHS) will be presented on page 109.

5.4.4. intchip4prhs - Interrupt Management Device

intchip4prhs device has two primary functions:

1. As PRHS processors only support a very limited number of external interrupts, a mechanism to manage a large number of interrupts of connected devices is required.
2. Each interrupt generating device should have an interrupt disable bit in one of his state machine registers. Additionally it might be beneficial, if dedicated interrupts can be masked out by the interrupt management system itself.

The *intchip4prhs* device offers the possibility to manage 32 incoming, active low interrupt signals and bundle them into one outgoing active-low interrupt signal. More than 32 interrupts can be handled by cascading several *intchip4prhs* devices.

The *intchip4prhs* device only implements a PRHS Bus interface and a register state machine as described in on page 64.

Register Interpretation

Each access to the *intchip4prhs* registers is interpreted as word access. Therefore, `icPRHSWidth` is ignored by the device.

interrupt status register (`GEN_BaseAddress`)

bit	R/W	description
[31:0]	R	Each bit represents the current status of the appropriate incoming interrupt with respect to the corresponding bit of the enable mask register. Bits have to be interpreted as active high.

interrupt enable register (`GEN_BaseAddress` + 0x4)

bit	R/W	description
[31:0]	R/W	Each bit enables the usage of the appropriate incoming interrupt. All interrupts are disabled (value '0') by default.

Access to any other register address than the above given, results in an unpredictable answer of the device.

Device generates interrupt, if any incoming interrupt signal is asserted ('0') and it's corresponding enable bit is set to '1'. No kind of interrupt prioritization is implemented in *intchip4prhs*.

Usage of this device by L4PRHS will be given on page 105.

5.4.5. timer4prhs - Timing Measurement in the PRHS Framework

The *timer4prhs* device provides the ability to measure time on a PRHS system. This includes a simple *"how much time has been elapsed since ..."* question and also the possibility to get a wake up call (in form of an interrupt) at a programmable time in future.

The basis for timing measurement is provided by a 32 bit wide soft-reset-able up counter, that increments its value at each rising edge of the system clock.

This counter is controlled via the *timer state machine*, which itself is controllable by a PRHS Bus attached register state machine.

Register Interpretation

Each access to the *intchip4prhs* registers is interpreted as word access. Therefore, `icPRHSWidth` is ignored by the device.

counter/threshold value register (`GEN_BaseAddress`)

bit	R/W	description
[31:0]	R	Current counter value (counter), interpreted as 32 bit unsigned integer.
[31:0]	W	Threshold value (threshold) for setting interrupt or wrap around to zero, interpreted as 32 bit unsigned integer.

status/control register (`GEN_BaseAddress` + 0x4)

bit	R/W	description
[31:4]	R	all bits are set to '0'
[3]	R/W	Interrupt Enable Bit (IntEn); Bit only controls signaling interrupt; it enables the timer to change to *INTERRUPT_PENDING* state whenever counter value reaches threshold value.
[2]	R/W	Timer Mode bit (mode); is set to '0' for *one-shot* mode, is set to '1' for *periodic* mode.
[1]	R	Timer Running bit (running); This bit is '1', if timer is in *IDLE* state, else '0'.
[1]	W	Start bit (start); If a '0' is written to this bit, timer enters *IDLE* state, if '1' is written timer is (re)started (state set to *START*) by soft-resetting the counter.
[0]	R	Interrupt Pending bit (IntPen); This bit is '1', if timer is in *INTERRUPT_PENDING* state, else '0'.

interrupt acknowledge register (`GEN_BaseAddress` + 0x8)

bit	R/W	description
[31:0]	R/W	Any access will acknowledge a pending interrupt. Timer then enters *IDLE* state if one-shot mode is selected. In case of periodic mode it restarts counting.

Access to any other register address than the above given, results in an unpredictable answer of the device.

Device generates interrupt, if interrupt enable bit is '1' and timer state machine is in *INTERRUPT_PENDING* state.

Timer State Machine

For better understanding of the functionality of the *timer4prhs* device and the interpretation of the bits of the register state machine, the state chart of the timer state machine is given in Figure 5.15.

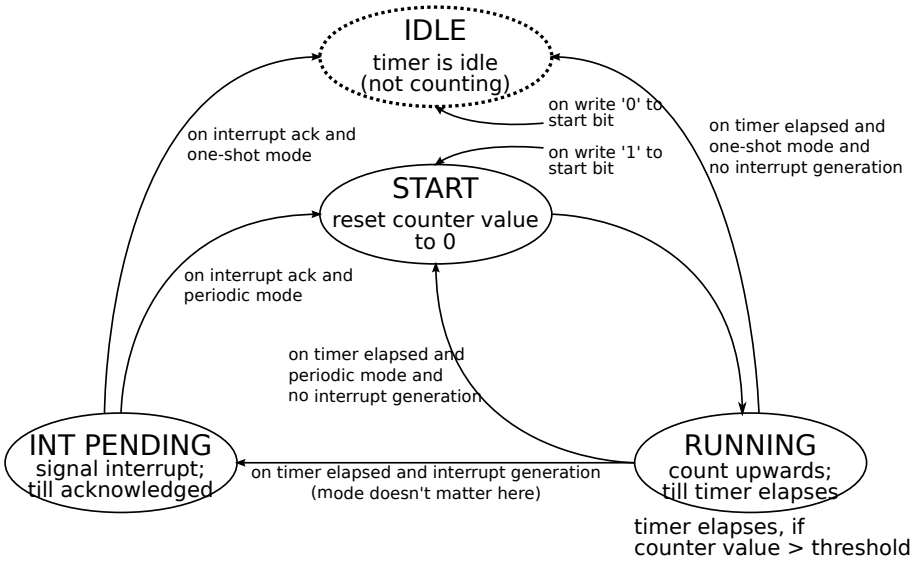


Figure 5.15.: Simplified state chart of the timer state machine.

Signal *sctimerelapsed* is set when counter value matches threshold value.

Usage of this device by L4PRHS will be presented on page 107.

5.4.6. uart4prhs - Basic User Interaction

The *uart4prhs* component connects an universal asynchronous receiver/transmitter (UART) to the PRHS Bus. It offers an easy and straightforward connection for user interaction over a serial line (RS232). The general structure of the *uart4prhs* component is given in Figure 5.16.

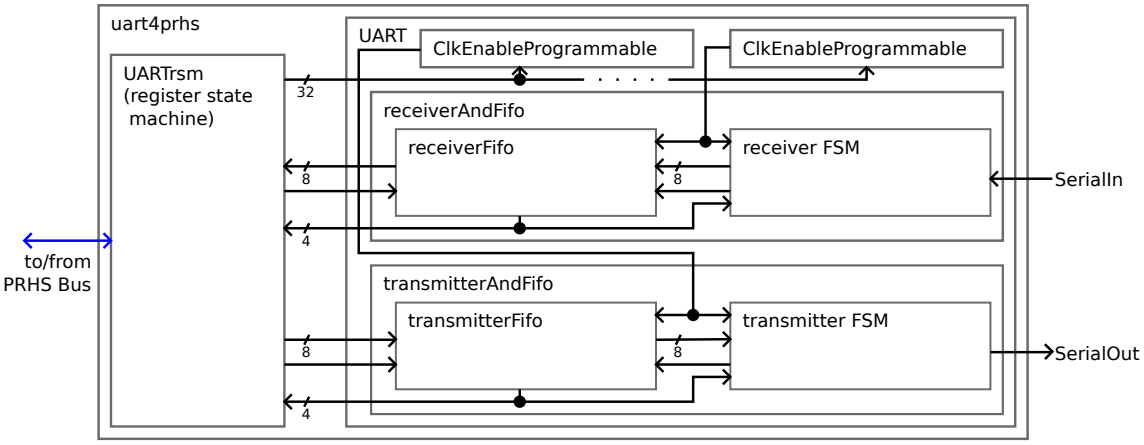


Figure 5.16.: General overview on uart4prhs device.

Interaction with the UART itself is done via a receiver and transmitter FIFO. Those FIFOs also implement the clock domain crossing, as the UART receiver and transmitter components work on low frequencies (refer to BAUD rate in RS232 specifications).

UART component is generating the needed BAUD rate by not using clock dividers but by using clock enable signals. This approach allows to use *uart4prhs* component in partial reconfigurable logic areas. Clock dividers would impose the need of taking care about clock buffer limitations regarding partial reconfigurable logic areas (see Xilinx partial reconfiguration user guide [Xil10b] for details).

Register Interpretation

Each access to the *uart4prhs* registers is interpreted as word access. Therefore, `icPRHSWidth` is ignored by the device.

receive/transmit register (`GEN_BaseAddress`)

bit	R/W	description
[31: 8]	R	all bits are set to '0'
[7: 0]	R	Read next received byte from receiver FIFO. If FIFO is empty, last received Byte is returned.
[7: 0]	W	Insert next Byte to send into transmit FIFO. If FIFO is full write is ignored.

status/control register (`GEN_BaseAddress` + 0x4)

bit	R/W	description
[31:13]	R	all bits are set to '0'
[12]	R/W	disable interrupt bit
[11: 8]	R	receiver FIFO status bits (empty, full, almost empty, almost full)
[7: 4]	R	transmitter FIFO status bits (empty, full, almost empty, almost full)
[3: 0]	R	all bits are set to '0'

baud rate generation register (`GEN_BaseAddress` + 0x8)

bit	R/W	description
[31: 0]	R/W	*value*is interpreted as unsigned integer to generate baud rate for receiver/transmitter of UART according to: $baud = \frac{<GEN\_SysClockinHz>}{value}$; initial value is $\frac{<GEN\_SysClockinHz>}{<GEN\_InitBaudrate>}$

Access to any other register address than the above given, results in an unpredictable answer of the device.

The device generates an interrupt, if the interrupt disable bit is '0' and the receiver FIFO is not empty.

Usage of this device by L4PRHS will be given on page 113.

5.4.7. bram4prhs - Block RAM based Memory

The *bram4prhs* component provides a bridge for attaching Block RAM to PRHS Bus. It combines two bRAM bridge state machines with a dual port Block RAM as given in Figure 5.17.

Bram4prhs resides on the secondary bus. Therefore, the bRAM bridge state machines have to handle non-immediate PRHS Bus transfers as first task. The second task is to translate the PRHS Bus address to the appropriate Block RAM address. The start address of the mapping of the memory address into the PRHS address space and the memory size itself are synthesis time configurable by generics.

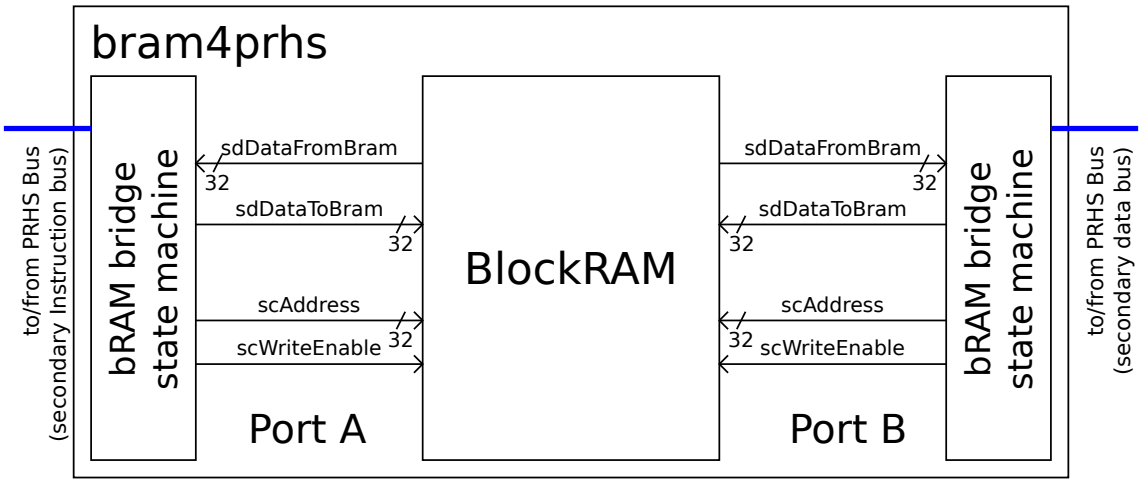


Figure 5.17.: Overview on bram4prhs device.

Primary purpose of bram4prhs is to hold the stage 1 boot-loader as described in section 7.2. For this reason, it is implemented as a dual port memory to connect it to the data and instruction PRHS Buses of a PRHS processor without the need for further (multiplexing) devices.

5.5. PRHS SD Bus Subsystem

External memory is required to provide enough memory to PRHS based systems. PRHS hardware has to support different platforms and therefore different types of external memory might be connected to a PRHS system. The following requirements are given for these connections:

1. The memory connection has to be generally independent of the used memory type. This offers flexibility and new memory types can be added easily.

2. Different PRHS hardware components should be able to access the external memory. A mechanism (interconnection system) to share an interface to the memory among those devices is needed.

The first requirement is fulfilled by defining a PRHS common, external memory access interface and protocol (*PRHS SD Bus*[5]), which is mapped to the appropriate protocol and interface of the board specific used external memory using a bridge. The second requirement is fulfilled by introducing a PRHS SD Bus specific interconnections system. These solutions results in an architecture, called PRHS SD Bus

[5]The term SD is used, because most of the platforms are usually shipped with SD-RAM. However, the external memory can also be of another type. PRHS SD Bus subsystem is designed to be also portable to other memory types.

subsystem as shown in Figure 5.18.

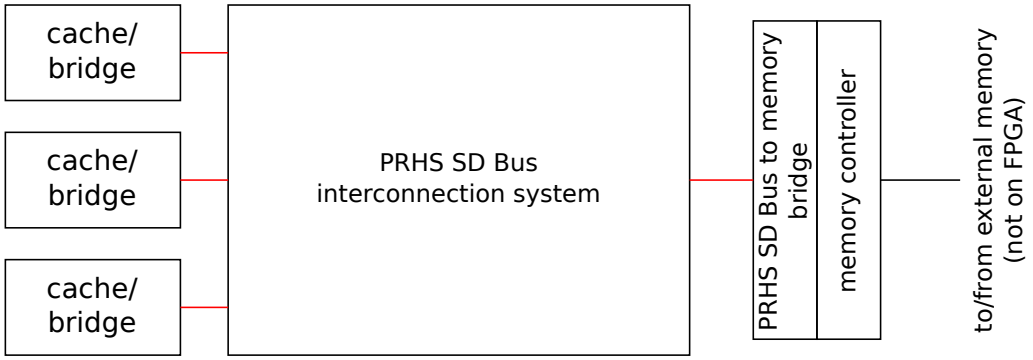


Figure 5.18.: PRHS SD Bus subsystem - general architecture.
This architectural scheme is an example using three cache/bridge devices communicating with the shared external memory.

The different aspects of the PRHS SD Bus subsystem are discussed in the following. First of all, the *PRHS SD Bus* is defined in section 5.5.1. Afterwards, the *PRHS Bus to PRHS SD Bus Cache* is presented in section 5.5.2. The *PRHS SD Bus interconnection system* is further explained in section 5.5.3. Finally, *PRHS SD Bus* bridges are presented in section 5.5.4.

5.5.1. PRHS SD Bus Definition

PRHS SD Bus Interface

PRHS SD Bus interface definition for Caches/Bridges, attachable to the PRHS SD Bus, is given in the following listing:

```
port (
  -- autogenerated interconnection system device ID
  idSDdevID         : in  idType;

  -- PRHS SD Bus request signals
  oeSDreqRequest    : out std_logic;
  ieSDreqAck        : in  std_logic;

  ocSDreqID         : out idType;
  ocSDreqAdr        : out std_logic_vector(31 downto 0);
  ocSDreqOperation  : out prhsBusOperation;
  odSDreqWData      : out prhsSDdata;
  ocSDreqWDataMask  : out prhsSDmask;

  -- PRHS SD Bus answer signals
  ieSDanswerValid   : in  std_logic;
  icSDanswerID      : in  idType;
```

```
18    idSDanswerData    : in  prhsSDdata;
);
```

Listing 5.3: Interface declaration for PRHS SD Bus.

using the self-defined data types:

```
type prhsBusOperation is (OPread, OPwrite, OPswap);
subtype idType  is std_logic_vector(4 downto 0);

subtype prhsSDdata is std_logic_vector((32 *(2 ** GEN_DSW))-1 downto 0);
subtype prhsSDmask is std_logic_vector((4 *(2 ** GEN_DSW))-1 downto 0);
```

PRHS SD Bus interface definition includes a generic, GEN_DSW. This generic is used for scaling the width of the data signals. As external memories differ in data width and optionally support burst transfers, GEN_DSW offers the possibility to exploit the performance benefits of those features. The width of the data signals is scaled, according to the following equation:

$$width_{\text{PRHS SD Bus data lines}} = 2^{GEN\_DSW} * width_{\text{PRHS Bus word}}$$

where $width_{\text{PRHS Bus word}} = 32$ as PRHS framework implements 32 bit systems.

PRHS SD Bus Protocol

The following table explains the meanings of the above given signals in detail.

Name	Description
idSDdevID	system-wide unique ID number for attached cache/memory; implicitly generated by the interconnection system
oeSDreqRequest	signal send by cache/memory to signal transfer of a request packet to attached interconnection system device (multiplexer or directly connected bridge block)
ieSDreqAck	acknowledge of attached interconnection system device to signal reception of request packet; acknowledge can be immediate (same cycle `oeSDreqRequest` is asserted) or non-immediate (see examples)
ocSDreqID	ID for tagging request packet; has to be equal to `idSDdevID`
ocSDreqAdr	32 bit wide address of request packet; address space is on a per byte basis, address of a request should be aligned (implemented by setting (2 + `GEN_DSW`)lowest bits of address to '0')

ocSDreqOperation	defines operation type of request packet; `OPread` for read, no answer packet will be sent by bridge block; `OPwrite` for write; `OPswap` for non-interruptible data exchange (read old value from, write new value to memory)
odSDreqWData	write data for request packet; content meaningless if `ocSDreqOperation` is set to `OPread`
ocSDreqWDataMask	mask appropriate bytes of `odSDreqWData` to exclude them from being written to memory ('1' means mask out, '0' means use for write)
ieSDanswerValid	sent by bridge block to signal validity of the other answer signals
icSDanswerID	ID of attached cache/memory who originally send the request, this answer is the response for
idSDanswerData	data that has been read by the bridge block from external memory

Write Transfers

A PRHS SD Bus write transfer example is given in Figure 5.19.

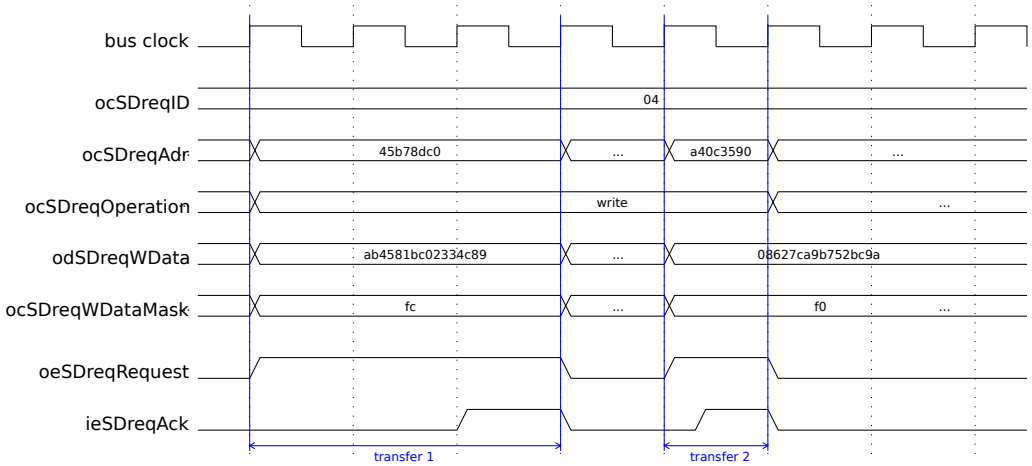


Figure 5.19.: PRHS SD Bus write example

In this example, `GEN_DSW` is 2 and `idSDdevID` is $0x04$. As write transfers do not have an answer packet send back from the bridge block, the answer signals of PRHS SD Bus are not included here.

Transfer 1 is a non-immediate transfer, whereas transfer 2 is an immediate one. Be aware of the necessity to not alter any request signals until a request is acknowledged. ***This is different to a PRHS Bus transfer.***

Read Transfers

An example read transfer and the appropriate answers is given in Figure 5.20.

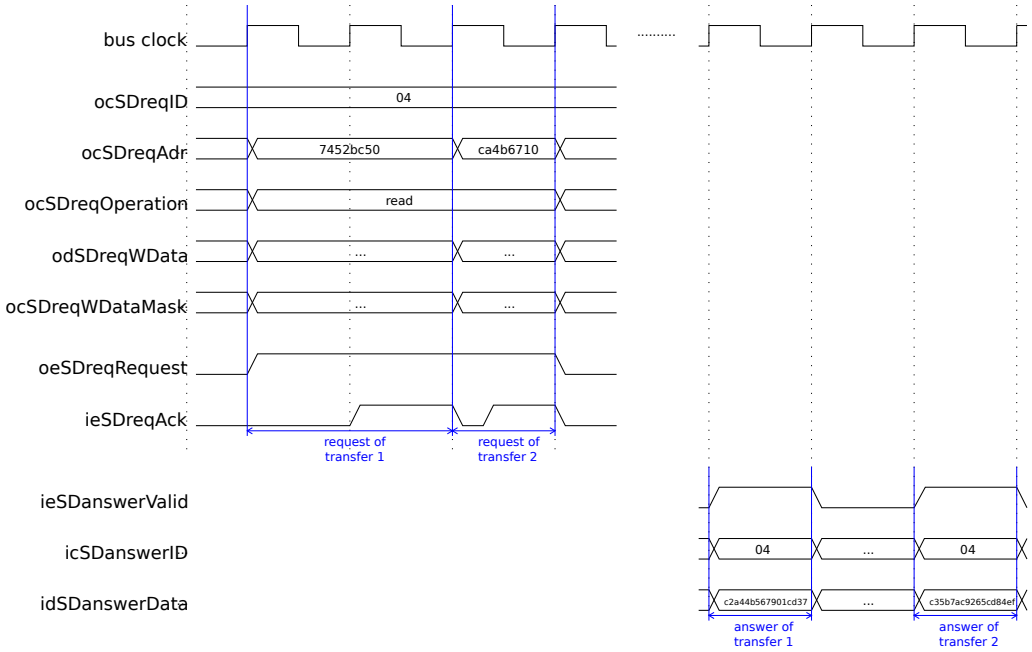


Figure 5.20.: PRHS SD Bus write example
In this example, `GEN_DSW` is 2 and `idSDdevID` is $0x04$.

The number of bus clock cycles between the acknowledgment of a request and receiving the appropriate answer (*latency*) is usually not predictable. It mainly depends on the used external memory and is slightly influenced by the depth of the PRHS SD bus interconnection system. Heavy load (a high number of request packets) on the interconnection system will also raise latency.

The order of answers given by the bridge block is the same order as the requests were issued by a cache/memory. All answers given by the bridge block are seen by all bridges/memories attached to the PRHS SD Bus.

Swap Transfers

Swap transfers are also possible with PRHS SD Bus. It combines an atomic read and write operation to the same address.

5.5.2. PRHS Bus to PRHS SD Bus Cache

The *PRHS Bus to PRHS SD Bus Cache* (cache) is the key component to translate *PRHS Bus* transfers to *PRHS SD Bus* transfers. Refer to page 52 for PRHS Bus definition and to page 74 for PRHS SD Bus definition.

Before presenting a brief overview on the implementation, necessary information regarding to data alignment is given.

Data Alignment Fundamentals

The data width of the *PRHS Bus* is limited to 32 bit ($= width_{\text{PRHS bus word}}$) and the data width of *RPHS SD Bus* is scalable by a generic GEN_DSW value according to the equation:

$$width_{\text{PRHS SD Bus data lines}} = 2^{GEN\_DSW} * width_{\text{PRHS Bus word}}$$

In addition both buses have an address signal included in their definition. The conversion mechanism of the *PRHS Bus to PRHS SD Bus Cache* between *PRHS Bus* and *PRHS SD Bus* is explained in the following. This is essential to understand where to find the appropriate place where data can be found. E.g. when it is necessary to implement a device, that also connects to the *PRHS SD Bus* interconnection system and accesses main memory data also accessed by the *PRHS Bus to PRHS SD Bus Cache*.

A *PRHS SD Bus* data word (named cache-line in the following) can hold multiple data words (32 bits wide, named data-word in the following) of *PRHS Bus*. How much of those data-words depends on the value of GEN_DSW according to the equation given above. Without loss of generality, the following examples use a value of 3 for GEN_DSW. Therefore, an example cache-line can hold 8 data-words.

Figure 5.21 shows how data-words are arranged in a cache-line and also presents the relation of the addresses used by the *PRHS Bus* and *PRHS SD Bus*.

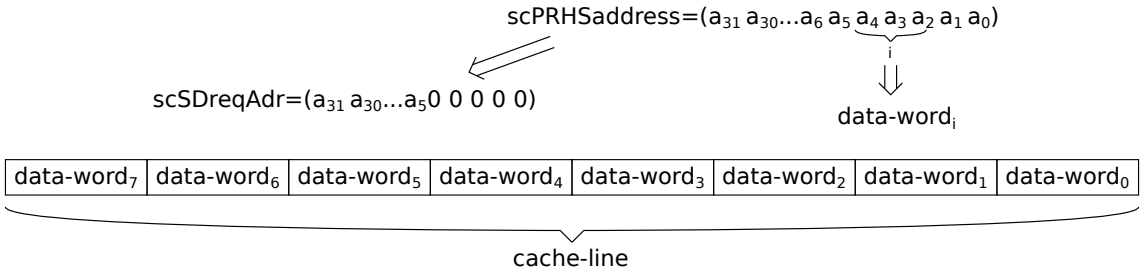


Figure 5.21.: Data-words arranged in a cache-line and address conversion.

The data-words are placed consecutively in a cache-line, starting with the least significant data-word on the left side. This follows a big endian principle and makes scaling the cache-line width easy to implement using GEN_DSW. Furthermore the *PRHS Bus* address (`scPRHSaddress`) is converted into the appropriate *PRHS SD Bus* address (`scSDreqAdr`). The original address bits a_1a_0 are set to zero (and should have been zero[6]) because *PRHS Bus* handles 4 byte words but addresses are byte oriented. The address bits preceding a_1a_0 can be used as an index i to identify data-word$_i$ in a cache-line. For the given example GEN_DSW is 3. These 3 bits are used to identify the index (formed by $a_4a_3a_2$).

Finding the right byte or halfword (16 bit) seems confusing at first glance, due to the fact, that PRHS processors implement little-endian data ordering. Figure 5.22 is used to give a better understanding for this example.

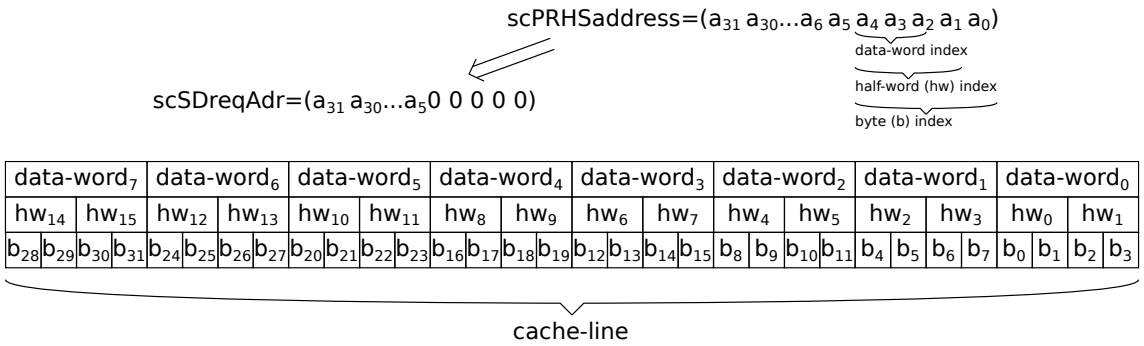


Figure 5.22.: Data-words, half-words and bytes arranged in a cache-line and address conversion.

Implementation Overview

As the requirements to such a central element like the *PRHS Bus to PRHS SD Bus Cache* may differ in terms of used area and performance (as this device might be a bottleneck for main memory access), three different architectures are implemented, *simple Bridge*, *buffered Bridge* and *Cache*.

Implementation details for the different architectures are given in the following. For a detailed introduction on Caches and the appropriate terminology see [HP07] and [PH09].

[6]Values other than zero indicate an alignment fault. Those faults have to be prevented/detected by the processor (or a coprocessor like a MMU) itself.

Simple Bridge

This implementation maps each PRHS Bus request to a corresponding PRHS SD Bus request. No caching mechanism is implemented at all. Therefore, the benefits of an immediate *PRHS Bus* answer cannot be achieved. Notably, performance benefits through pipelined processors cannot be achieved.

Nevertheless, this implementation can help to identify just as to proof cache coherency problems, as they cannot occur by design when using this implementation.

For the necessary conversion of addresses and data, this architecture implements a finite state machine as depicted in Figure 5.23.

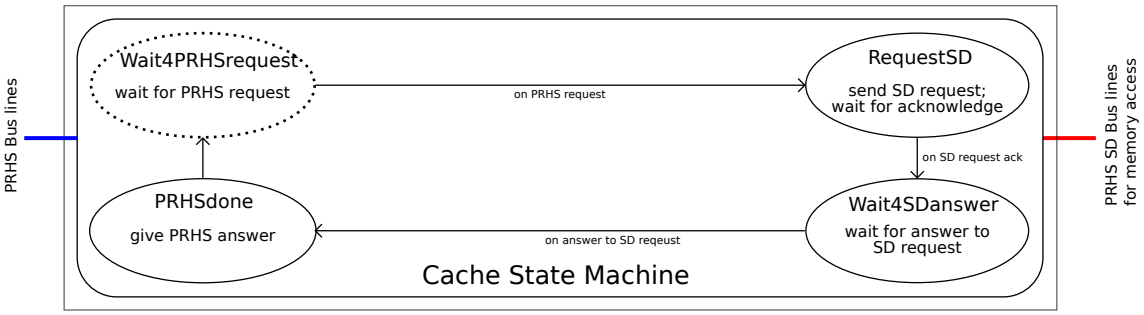


Figure 5.23.: Overview on simple bridge implementation.

Buffered Bridge

This implementation enhances the previous one by buffering (caching) the information about the previous PRHS SD Bus **read** request. If the incoming PRHS Bus read request would result in requesting the same PRHS SD Bus read request as the previous one, the request is not issued. By buffering the previous SD Bus read answer, an answer to the PRHS Bus request can be given immediately. A write or swap request is always send to main memory as PRHS SD Bus request. As the data of a PRHS SD Bus request is seen as a cache line, this implementation can be considered as a direct mapped, write through cache holding only a singleton cache line.. The enhancements results in the modification of the finite state machine as given in Figure 5.24.

Cache

This architecture implementation enhances the *buffered Bridge* implementation by adding a Block RAM based mechanism to cache more than on cache-line simulta-

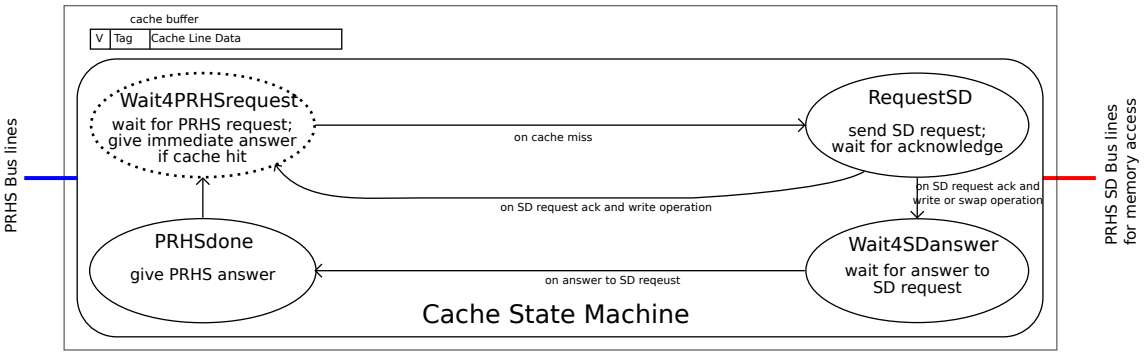


Figure 5.24.: Overview on buffered bridge implementation.

neously. The number of cached lines is configurable by a generic.

Block RAM lacks the problem[7], that it provides requested data only at the next rising(falling) edge of the system clock. Waiting an entire clock cycle to get the requested data from Block RAM would make immediate *PRHS Bus* transfers impossible. Therefore, the Block RAM only forms the second Stage of the caching mechanism, which takes two clock cycles for a read request. To support immediate *PRHS Bus* read transfers the buffering mechanism of the *buffered Bridge* implementation is used as the first cache stage.

As Block RAM contents are not "resettable", the *valid (V)* bits of the cache lines are not changed on reset. Therefore, an initialization sequence is necessary at reset (or startup) to set the valid bits of all cache lines to '0'.

Cache coherency is also a problem, occurring now. That is why this architecture also implements a *write invalidate protocol* in it's simplest form. A state machine is implemented, that "snoops" write transfers of another PRHS SD Bus component and invalidates it's own cache entry, if necessary to enforce a reload from main memory on the next read access to this cache line. This mechanisms ensures coherency among a data and an instruction cache attached to a PRHS-processor. The cache coherency protocol is a simplification of the MSI-coherence protocol (as described for example in [HP07].) It is more simpler than MSI as it has no *Modified* bit, because write through is implemented in it's pure sense and therefore doesn't require a modified bit for this implementation. In addition, it doesn't even use a *Shared* bit, as the protocol implicitly assumes a cache line as shared, even if it is only contained in one cache.

For this reason, the implemented cache coherence protocol can be interpreted as I-protocol. A cache line is either *invalid* or *valid*. This very simple protocol has been chosen as cache coherency is not in the scope of this thesis, but has to be ensured.

[7]See [Xil12a] for details on describing Block RAM in a high level language and synthesizable form.

For a detailed introduction to cache coherency protocols see [HP07].

Taking the above considerations as given, the finite state machine again requires modifications as given in Figure 5.25.

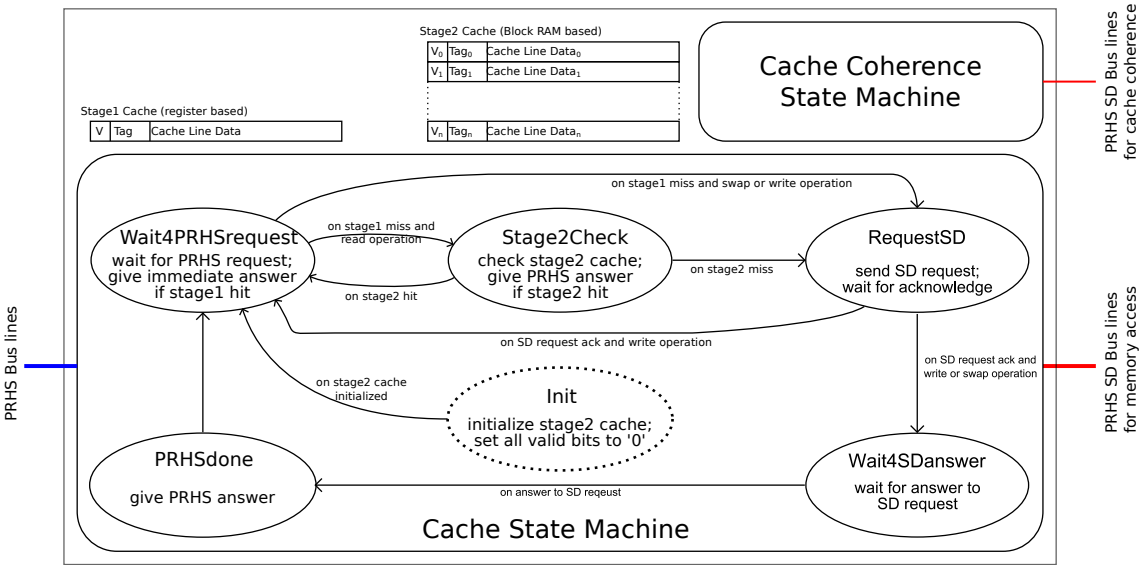


Figure 5.25.: Overview on cache implementation.

The current cache coherency protocol is suitable to build uniprocessor systems. It is sufficient to implement a virtualization system based on the main idea of this thesis. For the implementation of a true multiprocessor system that requires concurrent usage of shared main memory, a more sophisticated cache coherency mechanism needs to be implemented (e.g. MSI, MESI, or more complex ones). This might also require an adaptation of the entire PRHS SD Bus subsystem.

5.5.3. PRHS SD Bus Interconnection System

According to [RNP+] *The two dominant architectural choices for implementing communication fabrics for SoC's* are *transaction-based buses and packet-based Networks-on-Chip (NoC).*

Transaction based buses interconnect several devices via a shared or segmented bus line. A transfer between two devices is done on a transactional basis, meaning that the sender locks the bus for himself, performs a series of requests (which form the transaction) and then releases the bus. This is one of the main disadvantage of transaction based buses, because this bus lock prevents other devices from using the bus, which results in blocking those devices. Another big problem of transactional buses is their poor scalability [RNP+][GG00] regarding to the achievable frequency.

NoC's solve these transaction based bus problems by sending request and answer packets over a switchable network with routing functionality at the cost of area consumption. Another main disadvantage of NoC's is latency, because packets have to "hop" from on switch to another till the destination is reached.[GG00][BDM02].

For the PRHS SD Bus interconnection system scalability is a key factor. Therefore, a shared bus is not a suitable solution. The area consumption of a full blown Network on Chip would be excessive for the following reasons:

- A memory request is always directed from one of the attached caches/bridges to the one and only bridge block. This makes routing easy to implement.
- A memory answer is always directed from the bridge block to the requesting bridge/cache.

Hence, routing is only necessary from multiple sources to a singleton destination. Adding a unique requester ID to a memory request will allow broadcasting the corresponding answer to all caches/bridges, also including the original requester ID. The appropriate cache/bridge can now identify answers addressed to himself based on this ID.

Because of that, the resulting interconnection system solution is a combination of the shared bus (for the answers) and network (for the requests) concepts.

Scalability for the interconnections system is achieved by introducing the interconnection multiplexer. This device is used to build a cascading and scalable tree, with the root node connected to the PRHS SD Bus to memory bridge as given as example in Figure 5.26.

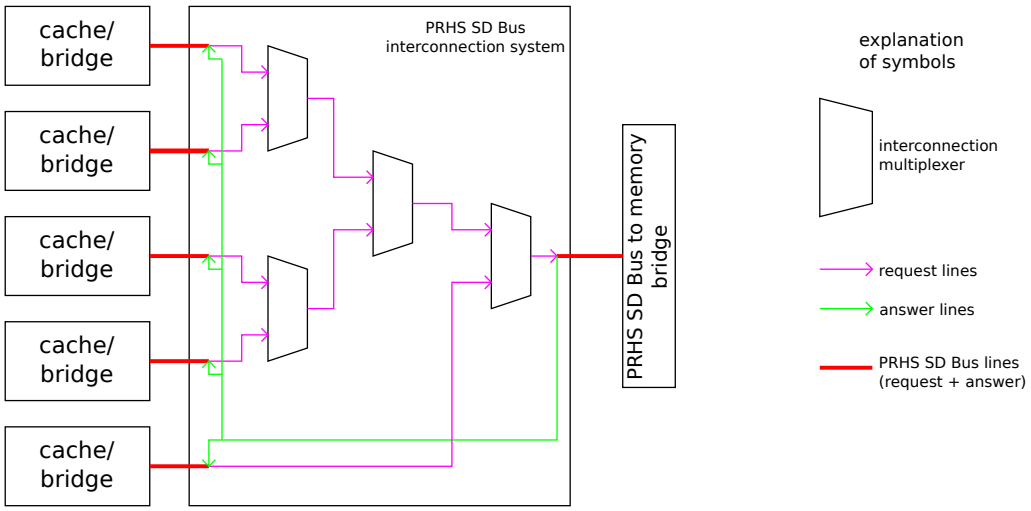


Figure 5.26.: Example for a PRHS SD Bus interconnection system.

The interconnection multiplexers ensure a fair serialization on the occurrence of

simultaneously arising requests on the inputs.

The answer lines of PRHS SD Bus are drawn outside the interconnection multiplexers for understanding reasons. For easier implementation needs, they are also included in the VHDL implementation of the interconnection multiplexers, but neither multiplexing nor same kind of "hopping" is implemented among them.

ID generation and assignment is implicitly done by the interconnection multiplexers. Necessary lines for this feature are not contained in Figure 5.26.

The following rules have to be taken into care, when implementing a new Cache/Bridge:

- Requests, send by the caches/bridges are "hopping" from one interconnection multiplexer to another. Therefore, the request is finished for a cache/bridge, if the interconnection multiplexer acknowledges a request although it has not been accomplished by the external memory.
- The tree architecture ensures that the order of request, issued by one of the caches/bridges, is obeyed.
- No cache coherency or synchronization principles are enforced by the interconnection system itself.

prhsSDBusMux - PRHS SD Bus Interconnection System Multiplexer

The general structure of the PRHS SD Bus interconnection system multiplexer (*prhsSDBusMux*) is given in Figure 5.27.

To understand the functionality of the prhsSDBusMux it is necessary to differentiate the PRHS SD Bus signals into request, answer and ID chain signals.

ID Chain Signals

The ID chain signals are for automatically generating and assigning device IDs to devices attached to the PRHS SD Bus interconnections system. The top level prhsSDBusMux gets ID 00000_2 and level 0 assigned as ID chain input. It will generate ID 00000_2 and level 1 as outgoing ID chain signals of Port A. For Port B this will be 00001_2 as ID and also 1 as level. In general, a prhsSDBusMux getting level i and ID $0 \ldots 00a_{i-1} \ldots a_0$ as input, will generate ID $0 \ldots 00a_{i-1} \ldots a_0$ for port A and ID $0 \ldots 01a_{i-1} \ldots a_0$ for port B. The level will be $i + 1$ for both ports. The width of the ID is actually limited to 5, which will result in a maximum number of $2^5 = 32$ devices to the PRHS SD interconnection system. A higher width can be easily implemented, but might result in high latency, as all attached devices share

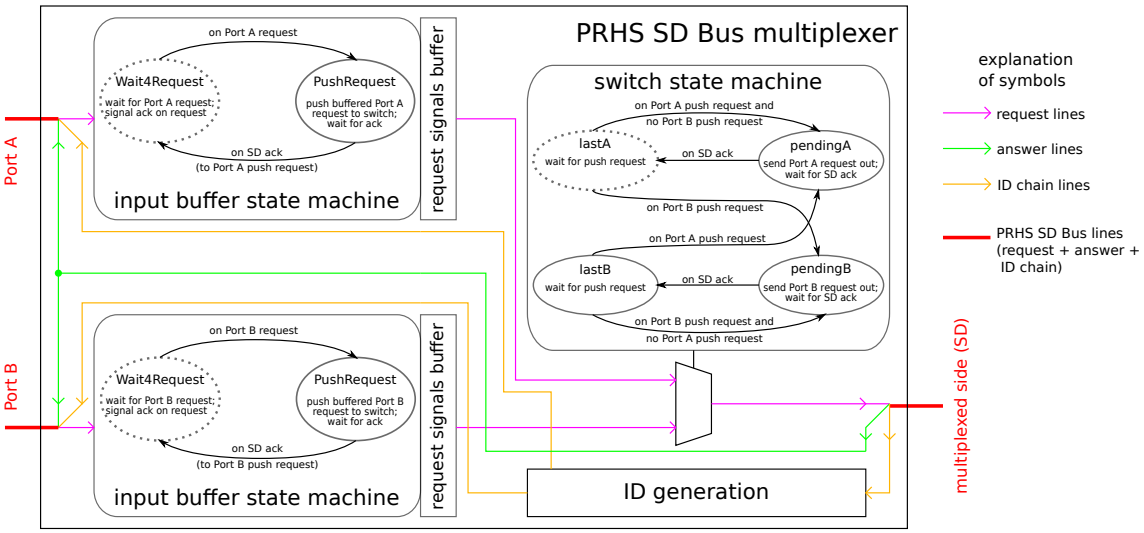


Figure 5.27.: Overview on prhsSDBusMux.

one singleton memory controller. The current width of 5 can be seen as a trade off value.

Answer Signals

As already discussed, all devices attached to the PRHS SD Bus interconnection system "see" all answers issued by the memory. Therefore, the prhsSDBusMux just propagates the answer signals to both ports without further investigation or buffering.

Request Signals

The request signals are the most interesting ones for the prhsSDBusMux. Each request issued by a device on one of the input ports is buffered and acknowledged (implemented by the input buffer state machine) first. So any attached device can progress with its work as fast as possible. The input buffer state machine then tries to push the request to the device connected at the multiplexed side of prhsSDBusMux (This is either the bridge block or an input port of another prhsSDBusMux). To ensure fairness and therefore enforce low latency for both devices, attached to the input ports, a switch state machine, whose state chart is given in Figure 5.27 is implemented.

5.5.4. PRHS SD Bus Bridges

For same external memory devices, a memory controller might be available as IP-core. Nevertheless, this memory controller and the PRHS SD Bus to memory bridge can be seen as a logical block, called *bridge block* in the remainder of this work. The tasks of this block are:

1. Map the common *PRHS SD Bus* protocol and interface to the external memory specific one, as already mentioned above.
2. Implement clock domain crossing issues, if the PRHS SD bus clock is different to the external memory interface clock.
3. Optionally implement some sort of buffering, if external memory latency is a bottleneck for devices (caches/bridges) attached to the interconnection system.

Detailed descriptions for board specific memory controllers and PRHS SD Bus to memory bridges can be found in the following.

ML505 PRHS SD Bus Bridge

The *ML505 PRHS SD Bus Bridge* translates PRHS SD Bus requests into requests for the DDR2 SDRAM memory controller as described in [Xil10a] (called *V5MC* in the following) and generated using *CoreGen*. It is used by PRHS framework on ML505 and XUPv5 Boards. The V5MC works on 64 bit words and implements a burst length of 4. Therefore, on ML505 and XUPv5 boards $width_{\text{PRHS SD Bus data lines}} = 64 * 4 = 256$ resulting in a GEN_DSW value of 3.

The *V5MC* comes with an user interface for issuing commands and appropriate data (command interface). It also provides an interface for data read from memory (read data interface). As both interfaces include asynchronous FIFOs, *ML505 PRHS SD Bus Bridge* doesn't has to care about clock domain crossing issues.

ML505 PRHS SD Bus Bridge is structured as shown in Figure 5.28.

PRHS SD Bus requests are handled by the *request state machine*. Besides acknowledging the request itself on the PRHS SD Bus, the *request state machine* translates the incoming read, write and swap request (PRHS SD Bus) into load and store operations of the *V5MC* command interface. A PRHS SD Bus read is transformed into a *V5MC* load, a PRHS SD Bus write into a *V5MC* store and a PRHS SD Bus swap into a *V5MC* load instantaneously followed by a store.

V5MC responds to load request on the command interface by giving the answer data after a variable number of clock cycles[8]. The ordering of the answers is strictly

[8] for a detailed discussion see [Xil10a]

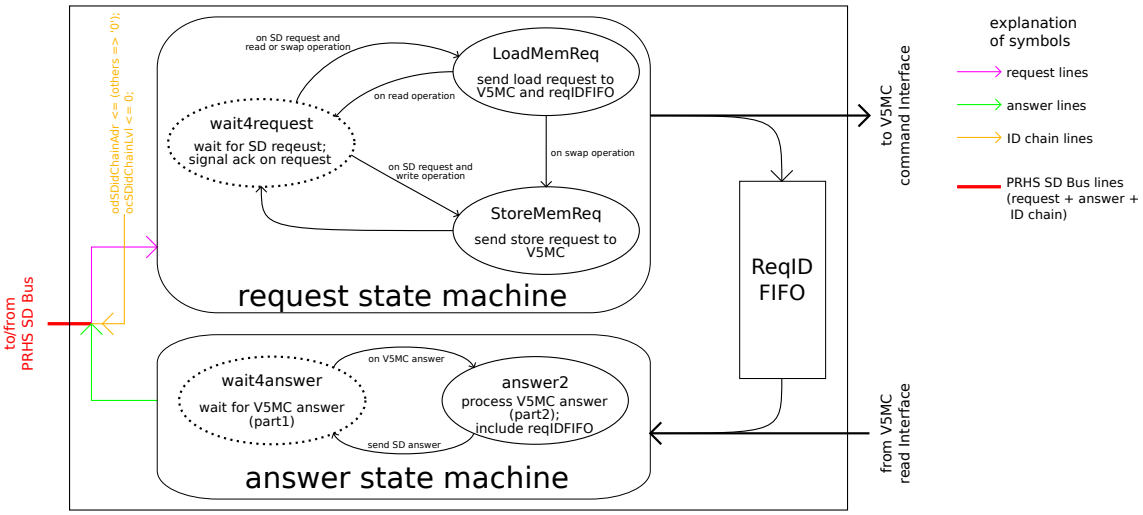


Figure 5.28.: Overview on ML505 PRHS SD Bus Bridge.

the same as the corresponding load commands.

PRHS SD Bus includes the requesters ID in an answer response. Therefore, the ID for requests which result in a PRHS SD Bus answer (load and swap requests) has to be buffered in the ReqID FIFO till the corresponding answer data is returned by the *V5MC*.

ML605 PRHS SD Bus Bridge

The *ML605 PRHS SD Bus Bridge* translates PRHS SD Bus request into requests for the DDR3 SDRAM memory controller as described in [Xil10c] (called *V6MC* in the following) and generated using *CoreGen*. It is used by PRHS framework on ML605 Boards. The V6MC works on 64 bit words and implements a burst length of 8. Therefore, on ML605 boards $width_{\text{PRHS SD Bus data lines}} = 64 * 8 = 512$ resulting in a GEN_DSW value of 4.

The *V6MC* comes with an user interface for issuing commands and appropriate data (command interface). It also provides an interface for data read from memory (read data interface). In contrast to the user interface provided for memory controllers for Virtex5 FPGAs (see previous section), both interfaces do not include FIFOs. Hence, *ML605 PRHS SD Bus Bridge* has to care about clock domain crossing

The resulting structure for the *ML605 PRHS SD Bus Bridge* is an extended version of the *ML505 PRHS SD Bus Bridge* as given in the previous section.

The structure of the *ML605 PRHS SD Bus Bridge* is presented in Figure 5.29.

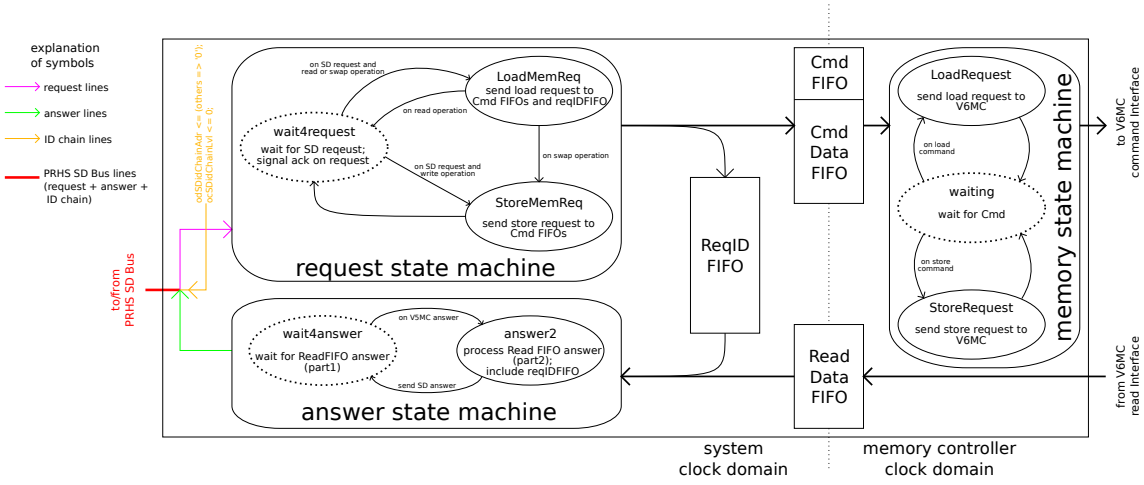


Figure 5.29.: Overview on ML605 PRHS SD Bus Bridge.
The state transition constraints and outputs are not given for readability reasons.

5.6. Platform Specific Devices

The devices presented in this section are platform specific. They can only be used, when an appropriate interface and/or controller is available on the used platform.

5.6.1. pstwo4prhs - Mouse and Keyboard Interfaces

The *pstwo4prhs* component connects PS2 protocoll based devices like keyboards and mice to the PRHS Bus. The general structure of the *pstwo4prhs* component is given in Figure 5.30. It is based on the works [Loo13] and [Las13].

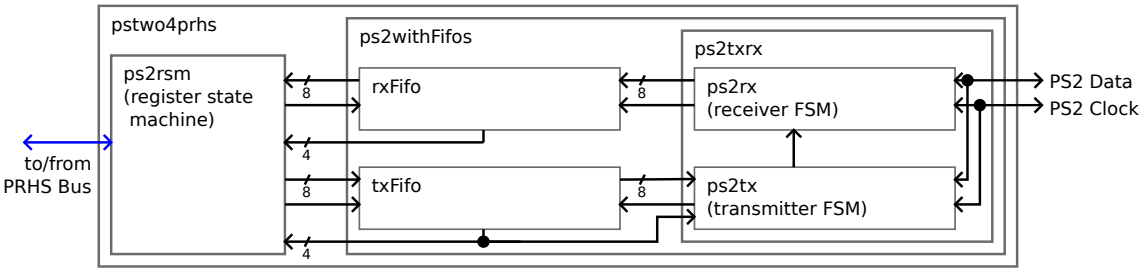


Figure 5.30.: Overview on pstwo4prhs device

Interaction with the PS2-transmitter-receiver device itself is done via a receiver and transmitter FIFO. All sub-devices of the *pstwo4prhs* component run on system clock. Hence, they can be used in partial reconfigurable areas.

Register Interpretation

Each access to the *pstwo4prhs* registers is interpreted as word access. Therefore, `icPRHSWidth` is ignored by the device.

receive/transmit register (`GEN_BaseAddress`)

bit	R/W	description
[31: 8]	R	all bits are set to '0'
[7: 0]	R	Read next received byte from receiver FIFO. If FIFO is empty, last received Byte is returned.
[7: 0]	W	Insert next Byte to send into transmit FIFO. If FIFO is full write is ignored.

status/control register (`GEN_BaseAddress` + 0x4)

bit	R/W	description
[31:9]	R	all bits are set to '0'
[8]	R/W	disable interrupt bit
[7:4]	R	receiver FIFO status bits (empty, full, almost empty, almost full)
[3:0]	R	transmitter FIFO status bits (empty, full, almost empty, almost full)

Access to any other register address than the above given, results in an unpredictable answer of the device.

If interrupt disable bit is '0' and receiver FIFO is not empty, the device will generate an interrupt.

Usage of this device by L4PRHS will be given on page 113.

5.6.2. sysace4prhs - Compact Flash Card Controller

The *sysace4prhs* device is based on the work of [Kab12]. It connects PRHS Framework with a CF-Card controller. This CF-Card controller is a dedicated chip (System ACE CompactFlash solution [Xil08]). *Sysace4prhs* device provides CF-Card based hard drive functionality on the boards providing a SystemAce chip (XUPv5, ML505, ML605).

As the system ACE controller device already implements a command/register based interface, the *sysace4prhs* device only has to bridge the PRHS Bus to the System ACE controller device.

Unfortunately the system ACE controller device is hardwired to a 33 MHz clock on the boards supported by PRHS framework. As PRHS system clock frequency isn't necessarily exactly 33 MHz. Therefore, the state machines, implementing the

bridging mechanism between PRHS data bus and system ACE controller device, also have to handle clock domain crossing issues.

For this reason the *sysace4prhs* state machine as described in [Kab12] has been split into two.[9] A (register) state machine, running on system clock, is connected to PRHS Bus and is responsible for implementing PRHS Bus handshaking. The SysAce (controller) state machine generates the signal sequences to interact with the system ACE controller device as defined in [Xil08] and runs on the required 33 MHz clock. Between those two state machines, a request-done handshaking mechanism is implemented. The corresponding control signals are used to solve clock domain crossing issues by implementing a two-flipflop synchronizer solution (see [**?**] for details).

The state machine dependencies and implementations are given in Figure 5.31.

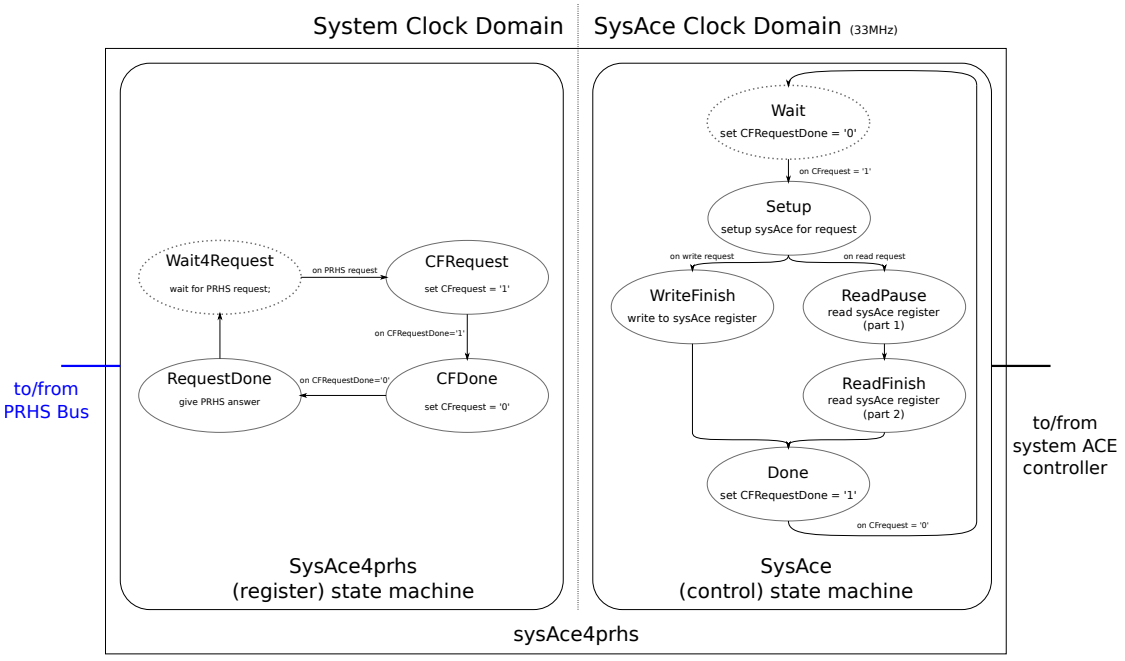


Figure 5.31.: State machines of sysace4prhs device.

Register Interpretation

System ACE controller device offers two access modes. The first is byte (8 bit) mode and the other word (16 bit) mode. As some boards (ML605) only support the 8 bit mode for wiring reasons, only 8 bit mode is supported by *sysace4prhs* device. Therefore, each PRHS Bus request is regarded as byte access. The registers

[9] The *sysace4prhs* device described in [Kab12] assumes the same 33MHz clock source for system ACE controller and PRHS Bus.

of System ACE controller device are directly mapped by the *sysace4prhs* device into the address space of PRHS processors. For this reason no register interpretation table is given here. See [Kab12] and especially [Xil08] for register interpretation of system ACE controller.

Device doesn't generate any interrupts.

Usage of this device by L4PRHS will be given on page 112.

5.6.3. v5emac4prhs - 10/100 Mbit Ethernet Controller for Virtex5 FPGAs

The *v5emac4prhs* has been added to PRHS framework by [Gr13]. It provides the connection to a 10/100 Mbit Ethernet controller for Virtex5 based boards (ML505 and XUPv5).

[Gr13] gives a detailed overview on the structure of *v5emac4prhs*. Therefore, only the *PRHS Bus* register interpretation is given here, which is necessary to understand the L4PRHS device driver implementation, which is not part of [Gr13].

Register Interpretation

Each access to the *intchip4prhs* registers is interpreted as word access. Therefore, `icPRHSWidth` is ignored by the device.

read Rx data register (`GEN_BaseAddress`)

bit	R/W	description
[31:10]	R	all bits are set to '0'
[9]	R	End of Frame bit (negative logic)
[8]	R	Start of Frame bit (negative logic)
[7:0]	R	data byte

write Tx data register (`GEN_BaseAddress + 0x1`)

bit	R/W	description
[31:10]	W	all bits are ignored
[9]	W	End of Frame bit (negative logic)
[8]	W	Start of Frame bit (negative logic)
[7:0]	W	data byte

interrupt disable register (`GEN_BaseAddress + 0x6`)

bit	R/W	description
[31:1]	R	all bits are set to '0'
[0]	R/W	interrupt disable bit

More registers are addressable regarding to [Gr13]. Only the given ones are used by L4PRHS.

Device generates interrupt, if Rx data Fifo is not empty and interrupt disable bit is not set.

Usage of this device by L4PRHS will be given on page 114.

5.7. Partial Reconfiguration Extension

The *partial reconfiguration extension* (PR extension) contains all devices to include dynamic and partial in-system reconfiguration capabilities into the PRHS framework.

Figure 5.32 gives an overview of the PR extension structure.

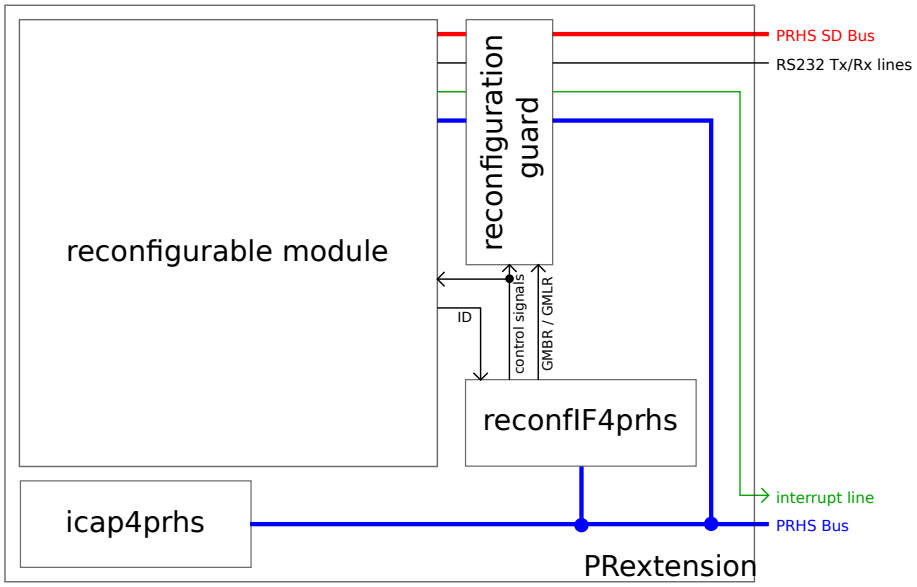


Figure 5.32.: Overview on PR extension.

The different components of PR extension are described in the following.

5.7.1. Reconfigurable Module

The *Reconfigurable Module* is a block box interface for instantiating a partial reconfigurable area, that can be used for dynamic and partial in-system reconfiguration at runtime.

The overall interface of the reconfigurable module is based on the requirements to a reconfigurable logic area of section 4.3:

PRHS Bus interface This allows to add a new device to the secondary data bus of the static system at runtime. This enables to dynamically exchange accelerator units at runtime.

PRHS SD Bus interface This allows any device/component/system currently instantiated in the reconfigurable module to access the system main memory.

Interrupt line This allows any device/component/system, instantiated at runtime in the reconfigurable area, to signal an interrupt to the static system. Interpretation of this interrupt is specific to the device/component/system currently instantiated in the reconfigurable module.

RS232 Tx/Rx lines This enables a RS232 based direct data exchange/user interaction with the device/component/system currently instantiated in the reconfigurable module, if necessary.

5.7.2. reconfIF4prhs - Reconfigurable Module Control Interface

The *reconfIF4prhs* provides a register state machine interface to the static system part to control the device/component/system currently instantiated in the reconfigurable module and the *reconfiguration guard* of the *PR extension*.

Register Interpretation Each access to the *reconfIF4prhs* registers is interpreted as word access. Therefore, `icPRHSWidth` is ignored by the device.

control register (`GEN_BaseAddress`)

bit	R/W	description
[31: 1]	R/W	ignored/unused
[0]	R/W	run reconfigurable module bit: This bit presents the inverted reset signal of the reconfigurable module. It also controls the *reconfiguration guard*.

module ID register (`GEN_BaseAddress + 0x4`)

bit	R/W	description
[31: 0]	R	Identifier of the device/component/system currently instantiated in the reconfigurable module.

guest memory base register (`GEN_BaseAddress + 0x8`)

bit	R/W	description
[31: 0]	R/W	Base register for main memory address space separation. See section 4.2.2 for details on address space separation.

guest memory limit register (`GEN_BaseAddress + 0xc`)

bit	R/W	description
[31: 0]	R/W	Limit register for main memory address space separation. See section 4.2.2 for details on address space separation.

Access to any other register address than the above given, results in an unpredictable answer of the device.

This register state machine doesn't generate interrupts.

Usage of *reconfIF4prhs* by L4PRHS is given on page 116.

5.7.3. Reconfiguration Guard - Address Space Separation and Sensitive Signals Gating

The *Reconfiguration Guard* (reconfGuard) has two tasks:

1. Gate sensitive signals: During an ongoing reconfiguration, performed via *icap4prhs*, the outgoing signals of the *reconfigurable module* might dangle. (See Xilinx PR user guide [Xil10b] for details.) To avoid, that those dangling signals affect other parts of the system, enable signals, the interrupt line and the RS232 transmit line of the *reconfigurable module* are gated with respect to the run reconfigurable module bit (see reconfIF4prhs above).

2. Implement address space separation between the static system part and the device/component/system currently instantiated in the reconfigurable module accessing main memory using PRHS SD Bus. This separation is done by using the guest memory base (GMBR) and guest memory limit register (GMLR), provided by *reconfIF4prhs*, as theoretically presented in section 4.2.2.

5.7.4. icap4prhs - internal configuration access port

The *icap4prhs* provides the ability to perform partial and dynamic configuration of parts of a FPGA by the FPGA itself. Therefore, a dedicated ICAP device provided by the FPGA is instantiated and connects to the *PRHS Bus* of the system. *The Internal Configuration Access Port (ICAP) is essentially an internal version of the SelectMAP interface. For more information, see the (Xilinx) family-specific Configuration User Guides.*[Xil10b]. The *icap4prhs* device only supports writing (for active reconfiguration) in 32 Bit mode. It doesn't provide any kind of read-back functionality to the PRHS framework (yet).

register interpretation

Each access to the *icap4prhs* registers is interpreted as word access. Therefore, `icPRHSWidth` is ignored by the device.

write to icap register (`GEN_BaseAddress`)

bit	R/W	description
[31: 0]	W	Data to write to ICAP. As PRHS implements little endian data ordering and ICAP needs big-endian, the bytes of the write-data are reordered. In addition, ICAP expects least significant bit of a byte on the left most position. Hence, the bytes also have to be mirrored bitwise

Access to any other register address than the above given, results in an unpredictable answer of the device.

Device doesn't generate interrupts.

Usage of this device by L4PRHS will be given on page 115.

5.8. Composed Hardware Modules

This section gives a brief overview of composed modules. Those modules purpose is to summarize functional blocks for easy reuse and adaptation.

5.8.1. base - Basic System

The *base* module composes all FPGA and *platform* independent devices, which are required to execute a full fledged operating system into a reusable module. It's general structure is given in Figure 5.33.

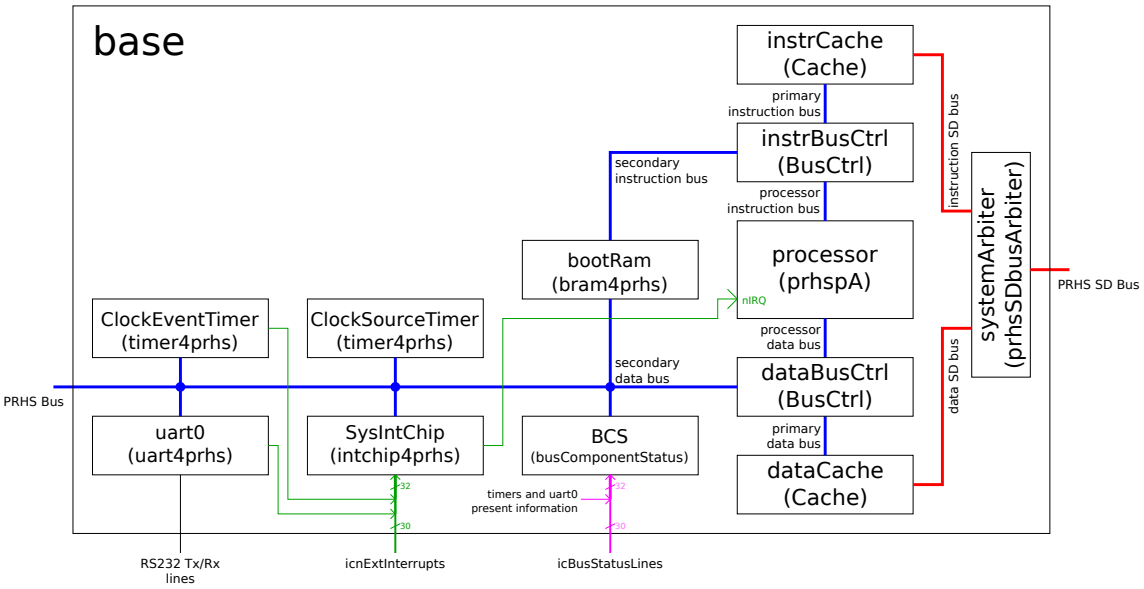


Figure 5.33.: Overview on composed module *base*.
For each device, the instance name is given (see source code). The module type is given in brackets.

It consists of:

- one processor and attached BusCtrl devices. One for the data bus and one for the instruction bus.
- an instruction and a data cache, attachable to the PRHS SD Bus interconnection system using a prhsSDbusArbiter.
- a bram4prhs which contains the first stage bootloader.
- an intChip4prhs for interrupt management.
- two timers, one for timer event generation (based on interrupts) and one for implementing continuous time measurement (clock source).
- a BusComponentStatus device to enable software to probe for devices attached to the secondary data bus.
- an UART for basic user interaction over a serial line.

It provides the possibility to add additional devices to the secondary data bus. This also includes the possibility to signal the presence of dedicated devices to the BusComponentStatus device and provide interrupts to the intChip4prhs device.

5.8.2. baseReconf - Extend Basic System with a PR extension Module

The *baseReconf* module combines a *base* module with an PR extension module to add partial reconfiguration capabilities (called "option reconf"). If a system is not build as a reconfigurable system, but as base system, the PR extension is replaced by a *uart4prhs* module (called "option base").

Figure 5.34 gives a schematic overview for *baseReconf* including both options.

5.8.3. baseReconfTop modules - Board Specific Top Modules

The *baseReconfTop* modules are the top level modules for the specific boards. In their simplest form, they only combine the *baseReconf* module with the appropriate board specific PRHS SD Bus Bridge and clock management devices.

Optionally, board specific devices can be added to the secondary data bus (PRHS Bus). Interrupt lines and *BusComponentStatus* lines for device probing are provided for these board specific devices.

As the structure of each *baseReconfTop* is platform specific, no figure is given here. See Appendix B for an overview on which devices are used on which platform.

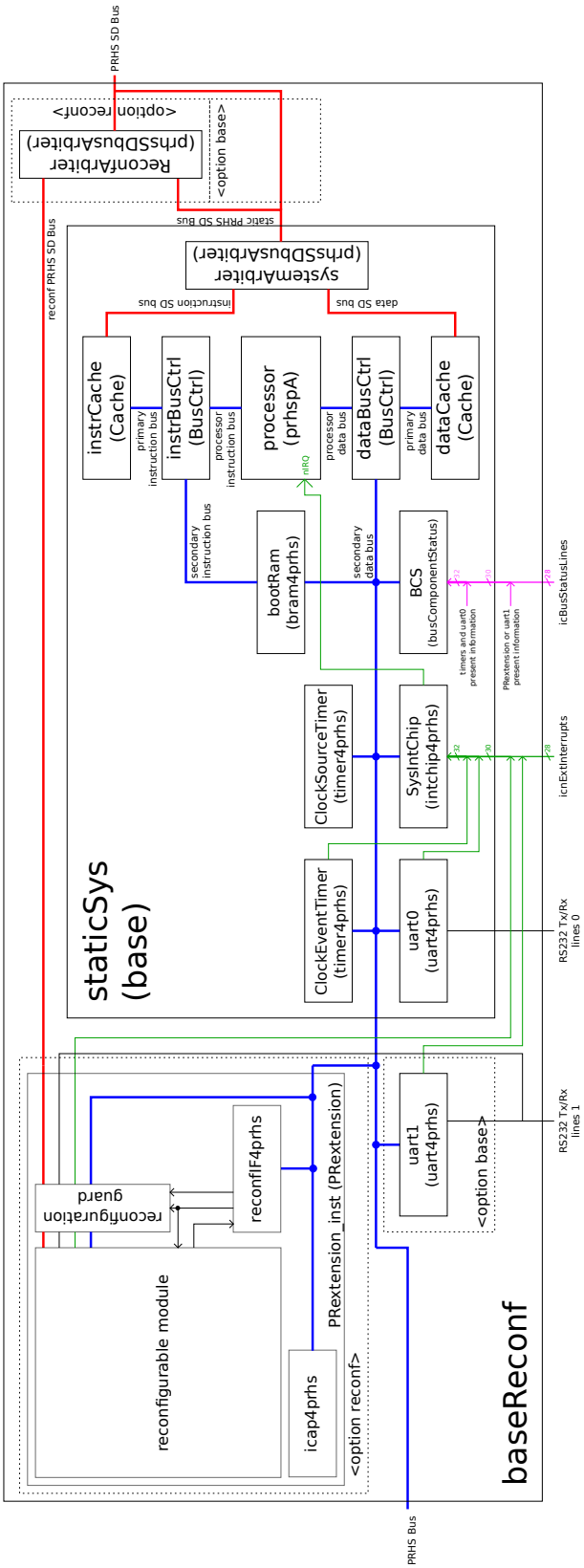


Figure 5.34.: Overview on composed module *basereconf*
For each device, the instance name is given (see source code). The module type is given in brackets.

6. PRHS Framework - Linux Kernel for PRHS - L4PRHS

As discussed in chapter 2, talking about (system) virtual machines includes talking about operating systems. In the previous chapter, the PPRHS framework hardware components have been presented. In this chapter, L4PRHS the operating system running on this hardware will be presented.

L4PRHS is based on Linux, as this is an operating system that is continuously developed further. It's main advantage is the availability of it's source code and therefore the possibility to adapt it for own needs, as done with L4PRHS.

Section 6.1 systematizes the different aspects of the adaption of Linux and where to find them in the sources. Section 6.2 presents processor specific adaptations of Linux kernel, 6.3 presents machine specific adaptations and 6.4 presents PRHS related device drivers in more detailed.

Version 3.8 of Linux has been used as starting point for adaptations, described in the following sections.

6.1. General Overview on L4PRHS

Figure 6.1 gives an simplified overview on a computer system running Linux as operating system. Operating system internals are shown more detailed than other components.

Main purpose of the kernel of an operating system is to *turn ugly hardware interfaces into beautiful abstractions* [Tan07], where beautiful abstractions is used as synonym for *System Call Interface*.

To achieve this, the kernel includes the device drivers layer, that maps interfaces of the kernel internal subsystems into appropriate device commands. For "conventional" devices, that fit into the categories *network*, *character* or *block* device, those device drivers are written in the high level language C. The corresponding device for the memory management and process management subsystem is the processor and/or memory management unit. Those "device drivers" are written in assembler,

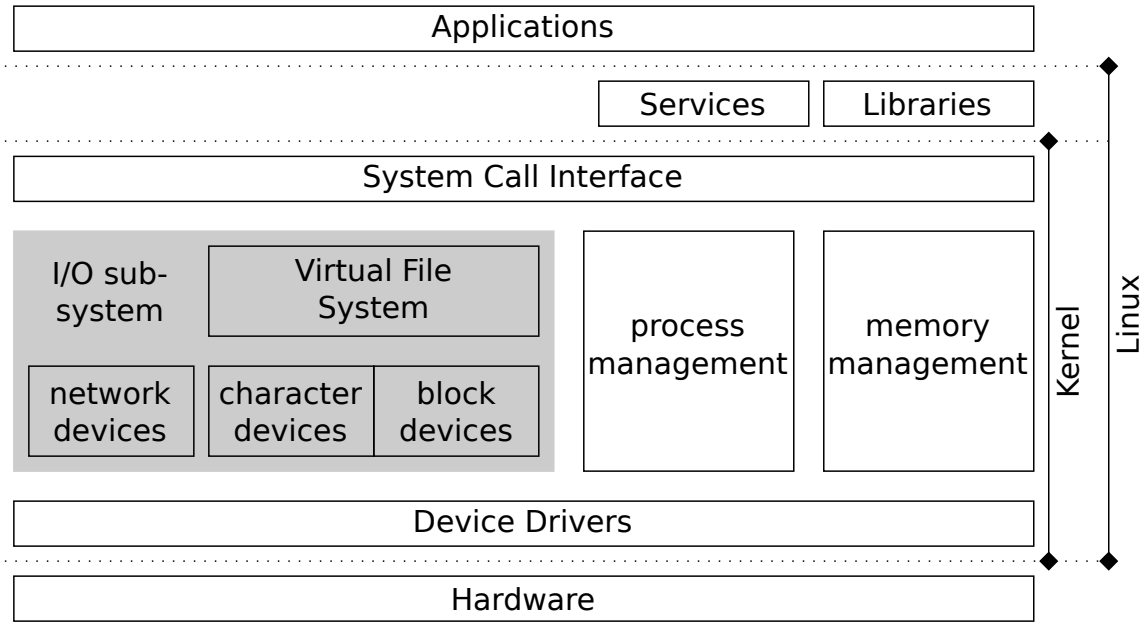


Figure 6.1.: Computer system running Linux.

as they have to be efficient for performance reasons. Additionally, some features are not expressible in a high level language (an atomic swap operation or a context switch are examples). Additionally, machine specific adaptations exist. They are based on "usual" devices, but are of fundamental importance for the overall functionality of a system. On the one hand this is the hardware based interrupt management. On the other hand, timers also fit into this category, as those are required to enforce preemptive scheduling.

Taking this differentiation as starting point, this chapter provides adaption details for "usual" devices in a separate chapter compared with the processor specific and machine specific adaptations.

6.2. Processor Specific Adaptations

As PRHSp-A is an ARM-ISA based processor, all kernel adaptations related to PRHSp-A are located in the *arch/arm* subdirectory of the Linux sources.

The ARM ISA is subject to an ongoing development. Therefore, the Linux kernel has to be adapted to the different ARM families and architecture versions. Those adaptations mainly focus on memory management, as there are processors without any memory management functionality, only a limited memory protection management (no virtual address space), or a full fledged memory management. Additionally, the on processor cache management is also subject of different implementations among

the different ARM families and architecture versions and therefore also subject to Linux adaptations. It's recommended to investigate some ARM processor manuals to find the differences in detail.

Processor specific adaptations are rarely document, neither in the kernel documentation nor in literature. Therefore, these adaptations are presented very detailed.

Defining PRHSpA as a New Processor

PRHSp-A is instruction set compatible with the arm8 ISA [Adv96] and therefore supports ARM architecture version 4 (ARMv4) . PRHSp-A implements no processor local cache, also managed by the memory management unit.

As this feature combination is not supported by any of the arm processors currently supported by Linux, PRHSp-A has to be added as a new processor in *mm/Kconfig*:

```
config CPU_PRHSPA
  bool "Support PRHSPA (ARM8) processor"
  select CPU_32v4
  select CPU_ABRT_NOMMU if !MMU
  select CPU_ABRT_EV4
  select CPU_PABRT_LEGACY
  select CPU_CACHE_V4
  select CPU_CP15
  select CPU_CP15_MMU if MMU
  select CPU_COPY_V4WT if MMU
  select CPU_TLB_V4WT if MMU
  help
    A 32-bit RISC microprocessor based on the ARM8 Instruction set.
    It has an onboard memory control unit but no onprocessor cache.

    Say Y if you want support for the PRHSpA processor.
    Otherwise, say N.
```

Listing 6.1: Adding PRHSpA as selectable option into Linux kernel.

MMU Usage Linux Kernel provides an option to configure kernel build for MMU-less systems. This is a fundamental decision when building a new kernel, because the decision MMU-less or not is a decision about whether taking advantage of/implementing a separate address space for each process (virtual memory) or not. For PRHS framework system, adapted Linux kernel provides both possibilities.

Developing new devices usually starts on the hardware side. MMU-less L4PRHS provides a fast and straightforward way to debug those new devices without the need to implement a basic device driver. This would be necessary on a MMU system to include the memory mapped I/O registers of the new device to be handled correctly by the MMU.

On Processor Cache The ARM subtree of Linux kernel expects each processor to have an on processor cache. As PRHSp-A doesn't implement on processor cache, two possibilities arise:

- Adapt the ARM subtree of Linux kernel to include processors with no on processor cache.
- Use an on processor caching mechanism supported by Linux Kernel, who's commands are accepted by PRHSp-A but don't execute them.

The second solution has been selected for L4PRHS, because this minimizes the need of Linux adaption.

Abort Handling Register indirect addressing instructions can produce MMU TLB misses resulting in an *abort* (a special type of interrupt). The different ARM families and architectures implement different strategies, whether or not and how the base register is modified in such a case. The right handling routines have to be selected for PRHSp-A, which keeps the base register unchanged in any case.

Instruction Set Architecture Allocation

To instruct the kernel build system to use the arm8 ISA for PRHSp-A based systems, it is necessary to add the appropriate information to *Makefile*:

```
tune-$(CONFIG_CPU_PRHSPA)       :=-mtune=arm8
```

MMU Handling for PRHSpA

Almost every ARM families and architectures introduced a new register interpretation for the system co-processor (co-processor number 15), which is responsible for holding the processor identifier, on processor cache management and MMU management. This requires a set of functions to be defined on a per processor bases in the ARM subtree of Linux.

The naming convention for those functions requires to add a new entry in *include/asm/glue-proc.h* to define a preamble on a per processor base:

```
#ifdef CONFIG_CPU_PRHSPA
# ifdef CPU_NAME
#  undef  MULTI_CPU
#  define MULTI_CPU
# else
#  define CPU_NAME cpu_prhspa
# endif
```

```
8 #endif
```

Listing 6.2: Adding PRHSpA prefix definition into Linux kernel.

Finally the system co-processor specific functions have to be implemented in assembler in *mm/proc-prhspa.S*. This file also includes the summarizing structure for all PRHSp-A specific function to be handed over to the kernel.

The following list gives a brief overview on the purpose of the processor specific co-processor functions (see *mm/proc-prhspa.S* for implementation details):

cpu_prhspa_proc_init called at boot time, when a processor is initialized, important for multiprocessor systems, unused for PRHSpA

cpu_prhspa_proc_fin called, whenever a processor is going to be shut down, important for multiprocessor systems, unused for PRHSpA

cpu_prhspa_do_idle called before a system is going to switch to idle loop (process with ID 0); This allows to take advantage processor specific functionalities for energy saving, e.g. reducing the processors clock rate. Unused for PRHSpA.

cpu_prhspa_dcache_clean_area called whenever kernel wants to clean areas of on processor cache, unused for PRHSpA

cpu_prhspa_switch_mm called, on every context switch to modify translation table base registers of the MMU, for PRHSpA, the TLB is entirely flushed.

cpu_prhspa_set_pte_ext called whenever a page table entry is to be created, wraps to a common function as PRHSpA implements a common page translation mechanism (ARM architecture specific)

cpu_prhspa_reset called whenever system is going to be reseted, has to set processor into a state, as if it had been physically reseted

Functions marked as unused need to be implemented, but return to caller immediately.

Finally the Linux build system has to be instructed to compile *mm/proc-prhspa.S* whenever PRHSp-A is used by modifying *mm/Makefile*:

```
obj-$(CONFIG_CPU_PRHSPA)        += proc-prhspa.o
```

6.3. Machine Specific Adaptations

After adding all required functionality to support PRHSp-A to the Linux kernel as presented in the previous section, it is now possible to add all information necessary to support a PRHS based system to Linux.

Adding a New Machine as Option to Kernel Build System

Firstly, a new machine has to be added to Linux kernel. As it utilizes PRHSp-A as processor, which is ARM ISA based, this takes place in the *arch/arm* subdirectory of the Linux sources.

A new sub-folder, named *proc-prhs* is added into *arch/arm* subdirectory. PRHS based systems are made available as option for the kernel by adding the following to *Kconfig*:

```
config ARCH_PRHS
  bool "PRHS based SoC"
  select CPU_PRHSPA
  select GENERIC_CLOCKEVENTS
  help
    Support for PRHS framework systems
```

Listing 6.3: Adding PRHS as system option into Linux kernel.

and place the appropriate information into the kernel build system by including the following in *Makefile*:

```
machine-$(CONFIG_ARCH_PRHS)             += prhs
```

Additionally an entry in the machine types database *tools/mach-types* is required. See this file for details.

Overview on Machine Specific Adaptations

To understand the functions to be provided by machine specific adaptions, the kernel booting sequence (for ARM based systems) is given in Figure 6.2.

The boot-loaders task is to load the kernel image to main memory and start executing it by setting the program counter to the start address of `stext()`, which is found at the start of the kernel image. The boot-loader is not part of the kernel boot sequence itself. Nevertheless, it has to provide information to the kernel. See chapter 7.2 for further details on PRHS boot-loader.

Kernel boot sequence starts with some ARM specific functions. The first one is to check if the used kernel is compatible with the used processor. The second function is only used by kernels using MMU functionality: Linux for ARM systems expect the kernel to reside on the upper part of the virtual address space. Main memory usually starts at real address 0. Before starting the kernel booting sequence itself, the MMU has to be set up by creating the appropriate page tables to map the kernel into the virtual address space it expects. Afterwards, the kernel booting sequence itself is started by calling the ISA independent kernel function `start_kernel`. The bootstrapping of essential operating system subsystems, with main effort on the

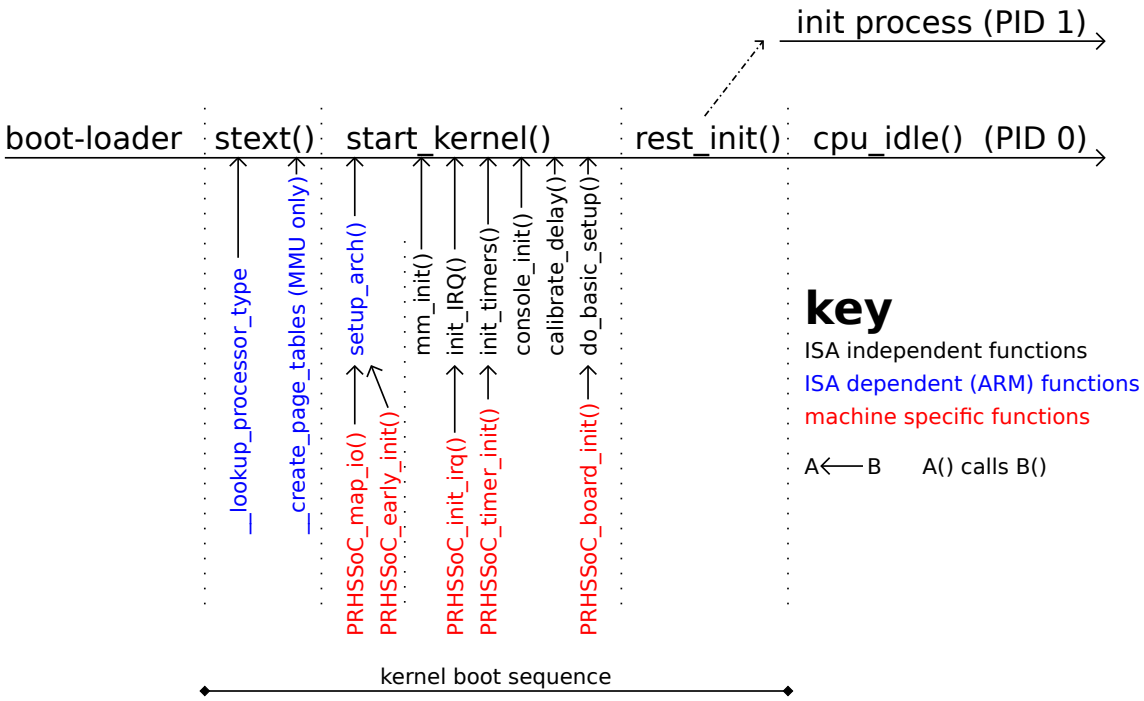


Figure 6.2.: Linux kernel boot sequence.

functions to be provided by machine specific adaptations are also given in Figure 6.2. Details are given in the following subsections.

At the end of `start_kernel`, a second process is created, the *init* process (Process ID 1). The `start_kernel` finally becomes the idle loop. The idle loop (PID 0, usually not seen on Linux system) is only executed, if no other process is ready to be scheduled. The *init* process (PID 1) is the first non-kernel functionality to be executed. At this point it is up to the program executed as *init* process to further boot the system. The current L4PRHS implementation for the *init* based boot is further explained in chapter 7.3.

The machine specific adaptations for PRHS based systems are summarized in the *mach-prhs* folder of the *arch/arm* subtree of the Linux sources as given in the following figure.

Interrupt Management

Interrupt management for PRHS systems is based on the *intChip4prhs* device presented at page 68.

Current PRHS hardware implementation supports up to 32 interrupt lines managed by one *intChip4prhs* device. This information has to be provided to the kernel by

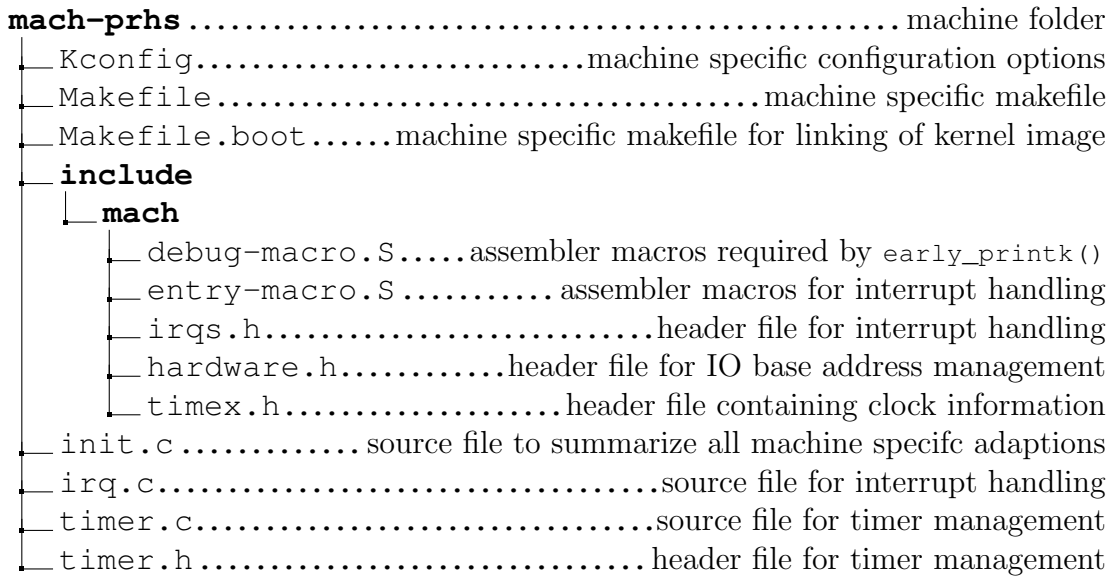


Figure 6.3.: Folder structure for machine specific adaptations.

defining in *include/mach/irqs.h*:

```
#define NR_IRQS  32
```

Changing this macro results in the necessity to rebuild the entire kernel, as this macro is of fundamental importance. Additionally the file *include/mach/irqs.h* contains macro definitions for naming dedicated interrupt line numbers.

At next, the kernel needs a mechanism to identify which device caused an interrupt, to hand the interrupt over to the responsible interrupt handler. This might include an prioritization mechanism. Machine specific kernel adaptations have to provide this mechanism by implementing an assembler macro in *include/mach/entry-macro.S* of the machine folder. On PRHS hardware this macro reads the *interrupt status register* of the *intChip4prhs* device. This register contains information about all pending (enabled) interrupts. Prioritization is implemented by choosing the "left most" pending interrupt to be handled.

Linux kernel provides a generic subsystem for IRQ handling. See *Documentation/-DocBook/genericirq* of the kernel sources for more details. For PRHS based systems, all 32 interrupt lines are handled using the *handle_level_irq* irq-flow method, because this fits the interrupt management philosophy of PRHS system (as described above) best. The IRQ subsystem expects to get a `struct irq_chip` filled with appropriate function callbacks as given in the following listing (for L4PRHS) for each interrupt line:

```
static struct irq_chip PRHSSoC_irq_chip = {
  .irq_mask       = PRHSSoC_int_mask,
  .irq_unmask     = PRHSSoC_int_unmask,
};
```

For L4PRHS all 32 interrupts use the same (above given) `struct irq_chip`. Whenever an interrupt occurs at a specific interrupt line the sequence given in Figure 6.4 is executed.

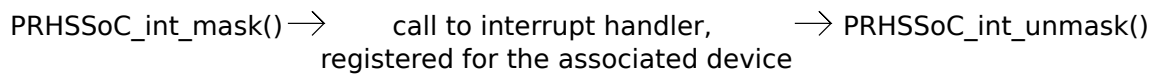


Figure 6.4.: Interrupt flow sequence.

The function `PRHSSoC_int_mask(x)` disables the signaling of an interrupt x by setting the appropriate bit of the *interrupt enable register* of the *intChip4prhs* device to '0'. This doesn't effect the interrupt enable/disable bit of the corresponding I/O-device.

Kernel is now able to enable interrupt signaling at all, so higher prioritized interrupts can be "seen". Afterwards it calls the interrupt specific handler of the device driver of the device associated with interrupt x.

Finally, kernel (re)enables the signaling of an interrupt by setting the appropriate bit of the *interrupt enable register* of the *intChip4prhs* device to '1' by calling `PRHSSoC_int_unmask()`. This doesn't effect the interrupt enable/disable bit of the corresponding I/O-device.

The last function implemented in *mach-prhs/irq.c* is `PRHSSoC_init_irq()`. Task of this function is to register `PRHSSoC_irq_chip` to all 32 interrupt lines. It is called, when kernel boot sequence initializes the interrupt subsystem (refer to Figure 6.2).

Timer Management

For a detailed discussion about the importance of timers for operating systems, see [Tan07] chapter 5.5 (page 386 et seq.). To enforce preemptive scheduling in an operating system, a mechanism to interrupt a user process after a given time is necessary. This is implemented using programmable timers. In Linux these are called *clock event devices*.

Programmable timers lack one problem: They are programmed to set an interrupt signal after a given time (in terms of timer clock ticks). If the interrupt occurs, the timer doesn't necessarily progress in counting timer ticks until it is reprogrammed/restarted. When only event timers are used for timing measurement, a system will "loose" time.

To avoid this time loss, Linux kernel also offers the possibility to use *clock source devices*. Those timers are started once, and count forward in time without interrupting. If kernel wants to know how much time has been elapsed since the start of

this source timer, it simply has to get the current counter value of the timer.

For this reason PRHS hardware implements two instance of the *timer4prhs* device (refer to page 70). One for usage as event timer, the other one as source timer. Both are implemented in *mach-prhs/timer.c* and *mach-prhs/timer.h*.

Event Timer PRHS event timer is instantiated by filling:

```
struct clock_event_device prhs_clock_event
```

with the appropriate data and callback functions. The callback function are `prhsTimer_set_mode()` and `prhsTimer_set_next_event()`. `prhsTimer_set_mode()` is called every time, when the kernel wants to change the mode of the event timer. `prhsTimer_set_next_event()` is called every time, when the kernel sets the event timer for the next event, it shall signal an interrupt. By implementing both functions, it is possible to select the *tickless system*[1] option for L4PRHS.

Additionally an interrupt handler for the event timer has to be instantiated:

```
struct irqaction  PRHSSoC_timer_irq
```

that is required to register the interrupt handler function `PRHSSoC_timer_interrupt()` to the kernel. This interrupt handler function has to inform the timer subsystem about the occurrence of the event timer interrupt and acknowledge the interrupt to hardware.

Source Timer PRHS source timer is instantiated by filling:

```
struct clocksource prhs_clocksource
```

with the appropriate data and callback function. The callback function is `prhs_read_cycles()`. It is called by kernel, whenever it needs the current value of the source timer counter to calculate the elapsed time precisely. Implementing a source timer allows to select the *High resolution timer support* option for L4PRHS. High resolution timer support is required, when the tick based time resolution (10ms or 1ms, depending on the *HZ* constant) is not sufficient.

Timer Initialization During kernel boot, also the timer subsystem is initialized (refer to Figure 6.2 on page 105). This includes calling the `PRHSSoC_timer_init()` function. Their task is to initialize the hardware timers and register the above presented timer structures to the kernel.

[1] On a tickless system, the timer doesn't interrupt periodically (Usually, Linux interrupts every 10ms or 1ms, depending on the *HZ* constant.). Instead, it is programmed to interrupt only when there will be something to do for the kernel (or more precisely the scheduler).

Machine Instantiation

Figure 6.2 on page 105 shows machine specific functions called during kernel boot sequence. All those function are bundled into a `struct machine_desc` (found in *arch/arm/include/am/mach/arch.h*) to be registered with Linux kernel for usage at kernel boot time.

The definition of the machine descriptor is done in *mach-prhs/init.c*. All functions of Figure 6.2 not presented in the previous sections, are defined there as well. Machine descriptor instantiation is implemented as:

```
MACHINE_START(PRHS_SOC, "PRHS␣SoC")
        .timer          = &PRHSSoC_timer,
        .map_io         = PRHSSoC_map_io,
        .init_irq       = PRHSSoC_init_irq,
        .init_early     = PRHSSoC_early_init,
        .init_machine   = PRHSSoC_board_init,
        .restart        = PRHSSoC_reset,
MACHINE_END
```

`MACHINE_START` and `MACHINE_END` are macros to include machine name resolution (`''PRHS SoC''`) and associated machine identifier (see *tools/machtypes* in *arch/arch* subsystem of kernel for details on machine names and identifiers).

`PRHSSoC_timer_init` is a function pointer to `PRHSSoC_timer_init`. This timer initialization function is described in the previous section.

`PRHSSoC_init_irq` is the interrupt subsystem initialization function as described above.

`PRHSSoC_map_io` is only used, when the memory management unit (virtual addresses) is utilized by L4PRHS. It registers the I/O device mappings to the memory management subsystem of the kernel for all system devices, not using a device driver. These system devices are both *timer4prhs* devices, *intChip4prhs* and *BusComponentStatus* device.

`PRHSSoC_early_init` checks for the used platform (FPGA and board) L4PRHS is currently running on and *printk*s it at kernel boot.

`PRHSSoC_board_init` checks *BusComponentStatus* device for platform type and attached devices, to register the appropriate `struct platform_device` instances to the platform bus subsystem. (See next section for details regarding platform bus subsystem.)

`PRHSSoC_reset` is called every time, a soft reset is initiatied by L4PRHS. As there is no special behavior at shut down for PRHS systems, this function calls the common architecture function `soft_restart(RESTART_ADDRESS);`. `RESTART_ADDRESS` is a macro defining the start address of stage 1 boot-loader in *bram4prhs* device.

6.4. PRHS Specific Device Drivers

For common I/O devices, Linux expects to handle them using a device driver. This allows Linux to present an abstract representation of hardware to applications. The device driver maps this abstract model to the needed command sequences of a dedicated device.

As this work wouldn't introduce device driver programming at all, only a brief overview for the device drivers implemented for PRHS systems is given. For a detailed introduction in Linux device driver programming see [CRKH05].

In subsection 6.4.1 a brief overview for understanding the principles of Linux device model is given. Afterwards the PRHS specific device drivers are given, differentiated as character, block and network drivers.

6.4.1. Overview on Linux Device Model

This section depends and uses information given in *Documentation/driver-model* file of the kernel sources and [CRKH05](chapter 14).

Devices, Drivers and Buses

The Linux Device Modell knows three primary elements:

Devices In Linux, each device is presented by a `struct device`. This struct hold all the necessary information to enable the kernel to interact with or to handle the device.

Drivers Device Drivers can export information and configuration variables, that are independent of any specific device. It's primary task is to handle multiple device instances and implement all the functions, that allow the kernel to interact with the devices.

Buses For Linux, a bus is a channel between the processor and one or more devices. For the purposes of the device model, all devices are connected via a bus, even it is only "virtual". A Bus is the connecting element between drivers and devices. Both register themselves to a bus but it's the task of the bus to inform the Driver about the change of a device (device is connected, or disconnected).

The general assumption of Linux is, that a bus is hot-pluggable and devices may be connected to or disconnected from the system at any time. This is the case for most of the common buses like PCI or USB today. Additionally, buses can be plugged into each other, like USB that is usually connected via PCI to the processor on x86

architectures.

Classes, Device Types and SysFS

Devices, Drivers and Buses can be used to model the physical (or a virtual) connection scheme for the devices present in a system. The class mechanism is a way to organize the zoo of devices by functionality.

User-space of Linux traditionally knows three different types of devices:

Character devices can be accessed as a stream of bytes like a conventional files.

Block devices can host a file-system. The underlying devices (e.g. disks) can only transfer entire blocks (usually 512 blocks of data today). Kernel provides an abstraction mechanism, that those devices can be handled like character devices from user-space, despite they have to be handled completely different inside the kernel.

Network devices are used to interact and communicate with other hosts on the basis of sending or receiving packets.

The SysFS (system filesystem) provides a way to get a filesystem view on the different ordering mechanisms. It is used by programs like *udev* or *mdev*, that automatically creates the file-system entries, associated with a device in the */dev/* directory of a Linux file-system (only block and character devices).

Platform Bus

For all devices not residing on a physically existing bus as expected by the kernel, a virtual bus exists: the *platform bus*. This bus is used by PRHS systems to get the device and driver information into the kernel. The `platform_device` instantiation and registration is done in the machine specific function `PRHSSoC_board_init` using the *BusComponentStatus* device as presented in the previous section.

Therefore, each PRHS device driver module registers itself as a `platform_driver` to the kernel, providing a `probe` callback function (called whenever a new device is registered). Selected devices also provide a `remove` callback function (called whenever a device is unregistered).

6.4.2. Block Device Drivers

Block devices can be accessed in Linux in two different ways:

raw by accessing the corresponding device file in */dev*

mounted by accessing files of a file-system, residing on a block device

Block devices have to register themself to the block layer subsystem, which is responsible for block I/O scheduling. The block I/O scheduling re-orders, merges and splits different requests to a block device for optimization reasons. The requests are handed over to the block device using a *block request queue* (`struct request_queue`) For a detailed overview on Linux block device drivers see [CRKH05] (chapter 16).

prhsace - Block Device Driver for sysace4prhs Device

This device driver was originally developed in [Kab12] and is located in *drivers/block/prhsace.c* of the kernel sources.

Device initialization is done with calling the `PRHSace_assign` function on a *platform_driver* probe. This function implements I/O address remapping (MMU related), allocates a `struct gendisk`, assigns major and minor number (241,0), sets the capacity and name (`PRHSaceCF`, used for naming under */dev*) to it and initializes the device specific *block request queue*. This includes registering the device driver specific request handling function(`reqfn()`). `Reqfn()` is invoked by Linux kernel every time a block has to be read or written. Finally, the disk representation is created with a call to `add_disk`.

Most work within prhsace driver is done in the `reqfn()` function. This function is called by the kernel, every time it wants the associated *sysace4prhs* device (see page 90) to perform read or write accesses to the inserted Compact Flash Card.

As the used Compact Flash Card contains the root file system, *prhsace* driver doesn't implement any kind of media exchange.

The current version of *prhsace* driver enhances the original version given in [Kab12] by adding the ability to retrieve the capacity of the inserted Compact Flash card and set the appropriate value in the device driver. In the original version *prhsace* driver assumed the CF Card to have a capacity of exactly 1GB resulting in occasional system crashes, when the inserted CF-Card had a smaller capacity and kernel tried to access a non-existing block.

6.4.3. Character Device Drivers

Character devices can be accessed as a stream of bytes like conventional files. For this purpose they are presented as character device file in */dev* in a Linux root file system. As devices of different vendors often have the same purpose, but differ in hardware details they can be summarized for abstraction purposes in subsystems

(also known as frameworks). The device drivers can take advantage of the common functionality of those subsystems and usually only have to implement the direct hardware access (read and write a byte, device configuration). Therefore, they are called Low-Level driver, whereas the framework that offer an abstract view to user space are often called high-level driver. Examples can be found in the PRHS related character device drivers given in this section.

prhs_uart Driver

The *prhs_uart* driver implements a low-level driver for the *serial* subsystem, which itself is part of the *tty* core subsystem. Additionally it implements the necessary functions for usage within the *console* subsystem of *tty* core. The driver is located in *drivers/tty/serial/prhs_uart.c* of the kernel sources.

The *prhs_uart* driver is designed to support multiple instances of the *uart4prhs* device (see page 71) as *serial* input and output interface. The first *uart4prhs* device is used also used as *console* device to see each kernel message printed using the `printk()` function.

prhs_ps2 Driver

The *prhs_ps2* driver implements a low-level driver for the *serio* subsystem, which itself is part of the *input* subsystem. The driver is located in *drivers/input/serio/prhs_ps2.c* of the kernel sources.

It was developed in [Las13] and [Loo13] to support multiple instances of *pstwo4prhs* device (see page 88). The *pstwo4prhs* devices are used to connect a keyboard and a mouse to the platforms were PS2 connectors are included and PRHS is running on. The differentiation between keyboard and mouse is done in the *serio* subsystem as the *prhs_ps2* driver is a low-level driver. Hence, the primary function of the driver is to implement the `ps2_write` and `ps2_rxint` functions.

`Ps2_write` is called by the *serio* subsystem, whenever it wants to send a byte to the attached PS2 device. `Ps2_rxint` is the interrupt handler, responsible for the attached PS2 device, to hand over a received byte to the *serio* subsystem.

Additionally a `ps2_close` and `ps2_open` function are implemented, to disable interrupt generation, if kernel isn't using the attached PS2 device and enable interrupt generation if it is used again.

6.4.4. Network Device Drivers

The role of a network interface within the system is similar to that of a mounted block device. A block device registers its disks and methods with the kernel, and then "transmits" and "receives" blocks on request, by means of its request function. Similarly a network interface must register itself within specific kernel data structures in order to be invoked, when packets are exchanged with the outside world [CRKH05].

There are differences between block and network devices. The first one is, for a block device, all communication is initiated by the kernel. (For this reason, the prhsace driver doesn't need an interrupt handler). In opposite, network communication usually can also be initiated by other hosts residing on a communication channel.

Additionally, network devices are not represented as a file in */dev* of the root file system like cahracter or block device. Network devices can only be found in */sys/-class/net* if *sysfs* is mounted on a system.

prhsEnet Driver

The *prhsEnet* Driver implements an Ethernet based low-level network driver. The driver is located in *drivers/net/ethernet/prhs/prhsenet.c* of the kernel sources. It handles the *v5emac4prhs* device (see page 91) described in [Gr13] (the *prhsEnet* driver is not part of [Gr13]).

At device probe time (call to platform bus probe function `PRHSenet_probe`) a new Ethernet network device is registered to the network subsystem. This includes the low-level driver callback functions as summarized in the following listing:

```
static struct net_device_ops prhs_netdev_ops = {
        .ndo_open               = prhsenet_open,
        .ndo_stop               = prhsenet_close,
        .ndo_start_xmit         = prhsenet_send,
        .ndo_set_mac_address    = prhsenet_set_mac_address,
        .ndo_tx_timeout         = prhsenet_tx_timeout,
};
```

`Prhsenet_open` is called, whenever the corresponding network device is *up*ped. This includes enabling interrupt generation and registering the interrupt handling routine `prhsenet_interrupt()` with the interrupt line of the associated *v5emac4prhs* device.

`Prhsenet_close` is called, whenever the corresponding network device is *down*ed. This includes disabling interrupt generation and unregistering the interrupt handling routine `prhsenet_interrupt()` with the interrupt line of the associated *v5emac4prhs* device.

`Prhsenet_set_mac_address` is called, when the kernel wants to change the MAC-

address of the network device.

`Prhsenet_send` and `prhsenet_tx_timeout` have to be regarded as a pair. `Prhsenet_send` is called, whenever the network interface shall send a packet. The network subsystem expects a device to signal success of transmission to the kernel. For *v5emac4prhs* device this is done by setting an interrupt. If the deivce isn't signaling the transmission done interrupt in a given time (defined at device probe time), a software timeout is generated, which results in calling `prhsenet_tx_timeout`.

`prhsenet_interrupt()`, registered during a call to `prhsenet_open`, is the interrupt handler for managing packet reception and transmission done signaling (see `prhsenet_tx_timeout`).

6.4.5. Partial Reconfiguration Extension related Device Drivers

The existence of a partial reconfiguration extension, implies the existence of two things: an *icap4prhs* device to perform in-system reconfiguration and a reconfigurable module with the appropriate interfaces for controlling the reconfigurable module. (See Figure 5.32 on page 92)

prhsicap Driver

This driver controls the *icap4prhs* device (see section 5.7.4), if a partial reconfiguration extension is available in the system. It is a straightforward character device driver, that is not part of any kernel driver subsystem. It contains the following file operations definition:

```
struct file_operations prhsicap_fops = {
        .open  =        prhsicap_open,
        .release =      prhsicap_release,
        .read =         prhsicap_read,
        .write =        prhsicap_write,
        .llseek =       noop_llseek,
};
```

The functions `prhsicap_open` and `prhsicap_release` are called, whenever the device file */dev/prhsicap0* is opened or closed. They are implemented in a way, to only allow exclusive access to the device, as writing different configuration streams simultaneously could result in a system crash.

For the same reason, seeking the device file (or more precisely the configuration stream) is prevented by binding the special `noop_llseek` function as seek callback function.

`Prhsicap_write` is implemented in a way, that allows to initiate an active reconfiguration by sending a reconfiguration stream (stored in the file *xy.bin*) to the ICAP by issuing the command:

```
echo xy.bin > /dev/prhsicap0
```

`Prhsicap_read` is implemented for completeness. It just returns a how-to banner whenever the icap device file is read, as reading from the underlying *icap4prhs* device is not implemented.

Reconfigurable Module Drivers

For the reconfigurable module, several device files are necessary, to represent the different available interfaces of the reconfigurable area: the control interface (*/dev/rm0ctrl*) and the SD-RAM interface including the real to physical mapping registers (*/dev/rm0sdmem*).

The **control interface** */dev/rm0ctrl* can be used to control the reset line of the reconfigurable area. For example, by issuing on the command line:

```
echo on > /dev/rm0ctrl            echo off > /dev/rm0ctrl
```

the reset is switched on and off. Reading from the control interface returns a status banner, containing the following information:

1. Current reset line status (on/off).
2. Identifier (32 bit) of the current configuration placed in the reconfigurable logic area.
3. GMBR and GMLR (physical addresses) of the physically contiguous memory area allocated for usage by the reconfigurable area. See address space separation related section 4.2.2 on page 30 for details.

The memory area defined by the GMBR and GMLR is accessible through the **SD-RAM interface** (*/dev/rm0sdmem*). This main memory area is shared between the host system and the system configured in the reconfigurable logic area. By issuing:

```
dd if=datafile of=/dev/rm0sdmem bs=4k seek=4
```

the data of the file *datafile* is written to main memory, accessible by the reconfigurable area starting at (real) address 16384 ($= 4k*4 = 4096*4$) of the reconfigurable logic area. For the host system, the corresponding real (and physical) address is GMBR + 16384. The shared memory area is allocated as contiguous DMA area. Therefore, it is part of the main memory, managed by L4PRHS. The size of the contiguous memory is configurable at driver module compile time.

It's configurable at kernel compile time, to implement */dev/rm0sdmem* either as block or as character device. Own experience shows, that the character device implementation allows faster access times, because block devices require more computational overhead in kernel. In opposite the block device implementation also has benefits. It allows to format a mountable file-system onto the underlying memory.

7. PRHS Framework - Software

7.1. C Compiler Toolchain

Software development and operating system compilation for a given system requires a compiler toolchain. The C compiler toolchain used in PRHS framework is based on gcc [FSF]. This section provides information, necessary to understand the difficulties of generating a system specific compiler toolchain and can serve as starting point for generating a new one. The information given in this section is based on [Bro] and [FSF].

A minimal C compiler toolchain has to provide the following components:

Compiler: The compiler translates source code into executable code for the targeted processor architecture.

Assembler: As essential parts of an operating system might be written in assembly language (for linux and L4PRHS this is the case), an assembler is required to convert it to processor specific bytecode.

Linker: The task of the linker is to combine one or several object-files into an executable program. The executable format depends on the targeted operating system.

Standard C library: Core C functionalities (like a simple *printf*) are provided by a standard library. This library (or to be more precise: libraries) provide object files to the compiler toolchain, which have to be compiled at toolchain generation time. They strongly depend on the targeted processor (used ISA) and operating system (system call interface, ABI version).

Assembler, linker and several other programs for manipulating binary executable files are summarized as *binary utilities* (binutils).

For Linux, operating system specific informations, required for the toolchain generation, are given in form of Linux header files which may differ slightly among Linux versions.

PRHS framework includes the PRHSp-A, which is based solely on the arm8 ISA. A lot of prebuild toolchains targeting Linux as operating system are available for

ARM processors. Choosing one of those prebuild toolchains is not possible for PRHS framework for the following reasons:

1. The toolchains don't support the most actual Linux version, as they are generated by using header files of older Linux versions. This may lead to strange errors, as the system call interfaces may differ slightly between different Linux versions.

2. The toolchains are targeting the most modern arm processors, which are not necessarily limited to the arm8 ISA. Newer ARM ISA version introduced new instructions and new arm processors also support additional ISAs like Thumb or Jazelle[1]. This is not a problem for compiling the Linux kernel, as gcc offers to change the targeted processor ISA at compile time. A problem occurs, when standard libraries are used within programs. The standard libraries are compiled at toolchain generation time and therefore might contain instructions, not implemented in PRHSp-A.

When generating a compiler toolchain, three different terms have to be taken care of:

Build This specifies the system, the compiler toolchain is build/generated on.

Host This specifies the system, the compiler toolchain shall be used on.

Target This specifies the system, the code, generated by the compiler toolchain shall be executed on.

As a consequence, many combinations are now possible:

1. Build=Host=Target: This is a native compiler, the common case for a compiler.

2. (Build=Host)≠Target: This is a cross-compiler. Executables for the target system are compiled on a development system. PRHS framework comes with a cross compiler toolchain.

3. Build≠(Host=Target): This is a cross-native compiler. The compiler toolchain is generated on a development system. The toolchain itself works on the same system it compiles executables for.

4. Build≠Host≠Target: This is called a Canadian Cross toolchain. It is the most complex and sophisticated way to generate a compiler toolchain.

More combinations are possible, but uncommon.

[1]Thumb and Jazelle are special ARM ISAs for generating smaller code. They present a subset of the classic ARM ISA and are translated/expanded in the decode stage of a processor.

7.2. PRHS Bootstrapping

In chapter 5 the components, constituting a system that can execute the L4PRHS operating system (see chapter 6) has been presented.

An interesting problem arises in how to mange the starting of executing L4PRHS, residing on a hard-disk. The task of loading an operating system from a hard-drive into the main memory and start executing it is called *bootstrapping*. The mixture of hardware and software, performing this task is called *boot loader*.

In this chapter, the bootstrapping process of PRHS framework will be presented. At first, the formatting of the hard disk, as expected by the PRHS boot loader is presented. Afterwards the two stages of the L4PRHS boot loader itself are briefly described.

7.2.1. Expected Hard Disk Structure

Nowadays two schemes for addressing data on disks is usual, **CHS** (cylinder-head-sector) addressing and **LBA** (logical block addressing). A brief introduction on the differences can be found in [Tan07](page 358 et seqq). This section uses LBA, where a disk is simply recognized as an array of data blocks, where the first block is at address 0. PRHS boot loader expects a block size of 512 bytes.

Figure 7.1 gives an overview on the disk layout, PRHS boot loader expects for the first blocks of a disk to operate correctly.

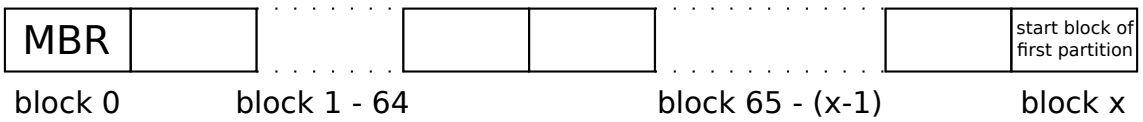


Figure 7.1.: Expected disk layout for boot loader.

For compatibility reasons (especially for *L4PRHS*), PRHS boot loader doesn't care about the first block (address 0). This block is called Master Boot Record (MBR) and usually holds the partition table and other information, which are important for an operating system working with file systems on base of disks.

Blocks 1 to 64 (holding 32kByte of data) is the data/program loaded by the first stage boot-loader into memory.

Blocks 65 to the first block of the first partition can contain data/programs used by the second stage boot-loader.

7.2.2. First Stage Boot Loader

The first stage boot-loader consecutively copies the data from blocks 1 to 64 of the disk to main memory starting at address 0. Afterwards, it sets the instruction pointer to memory address 0. This results in executing anything that was located in the disks blocks 1 to 64.

Those disk blocks usually hold the second stage boot-loader. Nevertheless, each program, smaller than 32 kbytes can be placed in those blocks and will be executed by PRHS after the first stage boot loader finished his job.

The first stage boot loader code itself is located in a Block RAM component (bram4prhs, see page 72) as initial content. Therefore, modification of the first stage boot loader requires re-synthesizing the hardware part of PRHS framework.[2]

7.2.3. Second Stage Boot Loader

The task of the second stage boot loader is to load the L4PRHS kernel image to memory and start executing it after setting the right register values as expected by L4PRHS (excerpt of *Documentation/arm/Booting* file of Linux kernel sources):

- register 0 has to be set to zero
- register 1 has to be set to 15000 (0x3a98); this is the machine identifier for PRHS based systems, can be found in *arch/arm/tools/mach-types* file of the Linux kernel sources
- register 2 has to contain the physical address of a tagged list (ATAG) or a device tree block (dtb). ATAG or dtb can be used by the boot loader to provide the Linux kernel with information about attached devices. PRHS boot loader sets register 2 to zero, because device discovering is managed by the L4PRHS kernel itself (see *PRHSSoC_board_init* function in section 6.3 on page 109).

The L4PRHS kernel image has to be placed consecutively on the disk, starting at block 65. Hence, the starting block of the first partition mustn't be placed on a low block address.

[2]Post-synthesis modification of Block RAM contents, as described in [Xil11], is not included in the PRHS framework design flow, yet.

7.3. System Base Software

Besides the kernel (L4PRHS) and libraries, specific applications are needed to get a fully usable operating system like Linux. These applications are summarized by the term *system base applications*. There is no clear definition among different Linux distributions, which applications belong into this category. For some essential tools like a shell, *ls*, *cd* or file system manipulating tools, there is no question, that they belong to the essential tools as they are usually summarized as *core utilities*. For other tools it's debatable, if they are of fundamental importance for running and using an operating system or not.

For Linux, adapted for usage with a PRHS based system, **busybox** [Vla] has been used to provide a fundamental set of system base applications.

Another straightforward possibility would be to compile the sources (self implemented or open source project based) for the needed base utilities entirely on your own. The most complex solution would be a L4PRHS specific application package manger, either newly implemented or an adapted version of an existing one.

8. Proof of Concept

In this chapter the proof of concept demonstrator for the main idea of this thesis is presented. Firstly, the necessary hardware components, developed for the proof of concept demonstrator, are presented. As next step, the software part of the proof of concept demonstrator, which represent the Virtual Machine Monitor is introduced. Finally, the result of the developed proof of concept demonstrator are evaluated with regard to the main idea.

8.1. Hardware

As overall hardware system, the top level hardware entity of the reconfiguration system, as presented in chapter 5, is used. For the proof of concept of the main idea of this thesis, the **reconfigurable module**, embedded in this reconfiguration system has to be used for system/device instantiation. Four different partial modules have been implemented for the proof of concept demonstrator as presented in the next subsections.

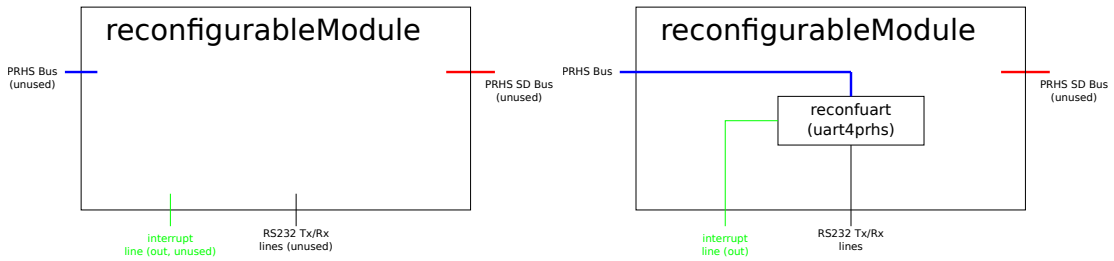


Figure 8.1.: Overview on idle and uart configuration.

8.1.1. Idle Configuration

The idle configuration sets the reconfigurable area into an "empty" state. All outgoing sensitive signals are tied to a value, that an activation of the reconfigurable area doesn't effect the static system part. This is the difference between the idle configuration and the *blank* configuration also generated by Xilinx partial reconfiguration design flow. In this blank configuration, the value of the outgoing sensitive

signals would be unpredictable. The activation of the blank reconfiguration could therefore affect the static system part.

8.1.2. UART Configuration

The UART configuration extends the static system part by another UART component. The UART component is connected to the static systems PRHS Bus, the reconfigurable area interrupt line and the Tx and Rx line of the RS232 interface of the reconfigurable area.

8.1.3. Core Configuration

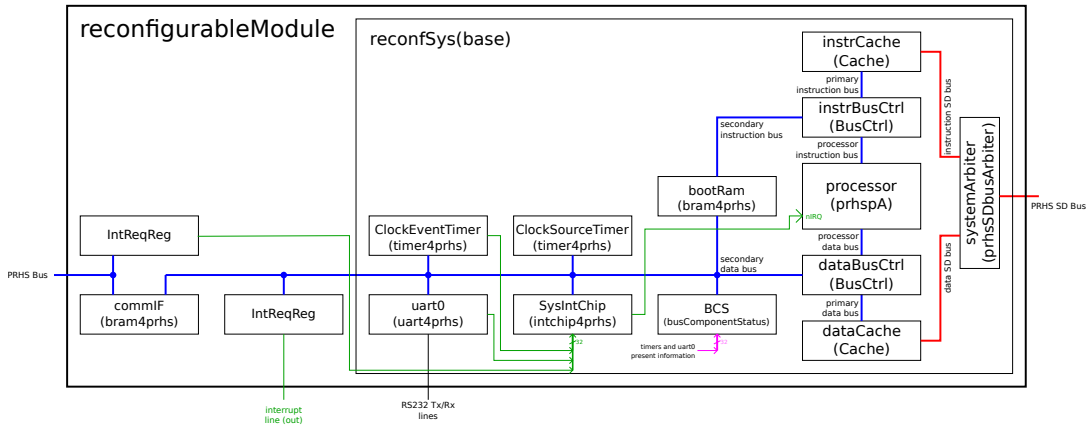


Figure 8.2.: Overview on core configuration.

This configuration is used to instantiate an additional machine, capable of executing an operating system. It instantiates a *base* system and extends it with devices (*commIF* and *IntReqReg*) to enable hardware supported emulation (see section 4.2.3).

The *commIF* is a *bram4prhs* instance. The dual port capability of *bram4prhs* allows to easily connect it to both, the guest and the host system simultaneously. The register interpretation is solely defined by the device drivers, as the device registers are just the memory cells of the Block RAM. There is one important requirement: The register interpretation of the guest system device driver and the host system emulation driver have to match.

IntReqReg - Interrupt Request Register Device

The *IntReqReg* device provides the possibility to the guest system (host system) to set an interrupt in the host system (guest system).

register interpretation Each access to the *IntReqReg* register is interpreted as word access. Therefore, `icPRHSWidth` is ignored by the device.

set interrupt register (`GEN_BaseAddress`)

bit	R/W	description
[31: 1]	R	all bits are set to '0'
[0]	R/W	Set interrupt bit. '1' sets an interrupt.

8.2. Device Drivers

8.2.1. Idle Configuration

The idle configuration doesn't need any further device drivers.

8.2.2. UART Configuration

The hardware instance of the *uart4prhs* device in the reconfigurable logic requires a device driver in the host operating system to work properly.

As *uart4prhs* devices are already used within L4PRHS, no additional driver is needed, but the additional *uart4prhs* instance has to be registered as additional device to the host operating system.

8.2.3. Core Configuration

The core configuration contains a hardware supported emulation interface. Based on this interface, hard disk functionality shall be provided to the guest machine.

This requires a block device driver included in the guest operating system and an emulation driver in the host operating system. Both drivers rely on the communication interface and the interrupt request registers. Inspect the drivers to find the interpretation of communication interface addresses in detail.

PRHShd - Guest System Block Device Driver

A block device driver for sysace4prhs device already included in L4PRHS is presented on page 112.

Block device drivers that are not part of a block device subsystem differ primarily in one central function. This is the `reqfn()` function. `Reqfn()` is invoked by Linux

kernel every time a block has to be read from or written to a disk.

Reasoned by this, the *PRHShd* looks almost the same as the sysace4prhs device driver. They differ in two points:

1. The capacity of the virtual hard disk is handed over from host to guest machine by using a register of the *commIF* device.

2. A block request (initialized by L4PRHS kernel with a call to the `reqfn()` function) is send from guest to host by also using the *commIF* device. This includes the direction of a request (read or write) and the block address of the block to be transfered. The transfer is initiated by the guest system by setting the associated interrupt line of the host system. Block data is also exchanged by using the *commIF* device. *PRHShd* driver uses polling to query for the end of a transfer.

Host System Emulation Driver

The task of the host system emulation driver is to transform the block transfer request of the guest system, issued by using the *commIF* device as explained in the previous section. The target of a block transfer is just a simple file on the host systems file system, holding the virtual disk information provided to the guest system. The host system emulation driver is controllable by the device file */dev/rm0vghd* on the host system.

8.3. Software

Virtual machine management is done by using the driver interfaces of the reconfigurable module area, which is presented in section 6.4.5.

The first step for setting up a new VM is to load the appropriate partial configuration stream by using the ICAP device (*/dev/prhsicap0*). Care has to be taken, that the reconfigurable area is in an active reset state (by using the control interface */dev/rm0ctrlif*). This is necessary to prevent the outgoing sensitive signals of the reconfigurable area from dangling during the reconfiguration process. The next steps for setting up the VM depend on the used configuration as explained in the following subsections.

8.3.1. Idle Configuration

As this configuration is not doing anything except setting the reconfigurable area into an "empty" state, the reconfigurable area can be switch on and off using the control interface */dev/rm0ctrlif*.

8.3.2. UART Configuration

To make the UART physically usable by the host system, the reconfiguration interface has to be switched on. As next step, the (host) operating system kernel has to be informed about the presence of this new device. As PRHS Bus doesn't support any auto-discover or hotplug functionality this has to be done in software. The most straightforward way is to load a kernel module, that registers the UART as new `platform_device`, as it is done for all static devices at kernel boot (see `PRHSSoC_board_init` function on page 109).

L4PRHS then will create an appropriate entry in `/dev` to use the configured UART.

To tear down the UART, the first step is to unregister the `platform_device`, associated with the configured UART. This is done by unloading the kernel module mentioned above. The final tear down step is to switch the reconfigurable area off, to physically disable the configured UART.

8.3.3. Core Configuration

For FPGA based system virtual machine investigations, this is the most interesting configuration. After the configurable area has been configured with the core configuration stream, the guest hardware is instantiated. Now the guest operating system has to be started. Therefore, the guest kernel image is placed at the required position in main memory.

A Linux kernel expects several processor registers set to defined values. Usually, this is done by the boot loader, before it starts executing the kernel boot image. For this reason an additional program (boot emulator), which sets the processor registers to an appropriate value, has to be loaded into main memory.

To provide the guest machine the illusion of accessing a hard disk, the host system emulation driver needs to be loaded. Afterwards the associated device needs to be instructed which (host system) file contains the hard disk information, presented to the guest machine.

After boot emulator and kernel Image are loaded into main memory and the host system emulation driver is initialized, the reconfiguration interface is started. In

consequence, the guest system is booting.

The guest system can be stopped by the host system by switching the reconfiguration interface off. The guest system will not shut down correctly in this case.

8.4. Proof of Concept in Comparison to Main Idea

The proof of concept demonstrator presented above, was implemented to show the following:

1. The proposed main idea of this thesis is applicable at all.
2. The usage of reconfigurable area to instantiate virtual machines is straightforward usable and combinable with the classical use of reconfigurable logic as additional device resource for a system.

To show the latter the UART configuration has been implemented. To show the first, the core configuration has been implemented. The core configuration implements selected features of FPGA based system virtual machines which are discussed theoretically in chapter 4. The practical implementation of those features in the proof of concept demonstrator and the results are presented in the following.

8.4.1. Processor related Results

Due to the lack of a second processor ISA in Partial Reconfigurable Heterogeneous System (PRHS) framework, guest and host machines processor use the same ISA. For this reason guest machine and host machine are the same from the architectural perspective.

So the proof of concept demonstrator doesn't implement the possibility of instantiating a guest system, who's ISA isn't provided by emulation or binary translation.

However, the proof of concept demonstrator implementation allows to show another benefit of reconfigurable logic based system virtual machines compared to conventional virtual machines: the virtualization mechanisms result in a computational performance loss for the guest system in comparison to the architectural same host system (see page 43 for a brief computational performance discussion).

For performance comparison between guest and host system, two benchmarks are used:

1. The BogoMips benchmark is compiled into the Linux kernel. At kernel boot time, the BogoMips number of the system is calculated. It can be obtained

by reading from the */proc/cpuinfo* file.

2. The Dhrystone benchmark [Wei84], which is available as an open source software.

Both benchmarks are regarded as suitable for comparing the computational performance of systems based on the same architecture, but not for comparing systems based on different architectures.

BogoMIPS values are identical (39.73 BogoMips[1]) for guest and host system.

For the Dhrystone benchmark different values are measured according to the following methods: Five different values were measured. $P_{host,off}$ is the result for the host system with reconfigurable area switched off. $P_{host,idle}$ is the result for the host system with reconfigurable area switched on, but an idle guest system.[2] $P_{guest,idle}$ is the same for the benchmark running on the guest system with an idle host system. $P_{host,both}$ and $P_{guest,both}$ are the performance values for a simultaneous benchmark run on both, host and guest system. Relative values are given here, because the discussion has to focus on performance differences between the host and the guest system. The peak performance (100 %) was achieved with the host system running and guest (reconfigurable area) switched off. The results are:

$$
\begin{aligned}
P_{host,off} &= 100\% \\
P_{host,idle} &= 99.5\% \quad \text{related to} \quad P_{host,off} \\
P_{guest,idle} &= 99.5\% \quad \text{related to} \quad P_{host,off} \\
P_{host,both} &= 98.7\% \quad \text{related to} \quad P_{host,off} \\
P_{guest,both} &= 98.8\% \quad \text{related to} \quad P_{host,off}
\end{aligned}
$$

Comparing $P_{host,idle}$ with $P_{guest,idle}$ and $P_{host,both}$ with $P_{guest,both}$ shows, that the guest system and the host system performances are identical. So there is no computational performance loss for the virtual machine compared to the host machine.

The reason for the lesser performance values of $P_{host,idle}$, $P_{guest,idle}$, $P_{host,both}$ and $P_{guest,both}$ is the shared memory controller. If both, the host and the guest system try to access main memory simultaneously, one of the systems has to wait. To emphasize this, the cache and memory performance for the proof of concept demonstrator has also been measured, using the ramspeed benchmark [HB09]. This benchmark measures the read and write cache/memory access performance in MegaBytes per second based on an increasing blocksize for reads and writes.

The results are shown in Figure 8.3. $R(x, y)$ is the read performance, $W(x, y)$ is the write performance. Performance is again given as a relative value. The reference

[1] System runs at 100MHz.

[2] Idle means, that the operating system only receives a timer interrupt every 10ms and is waiting for shell input

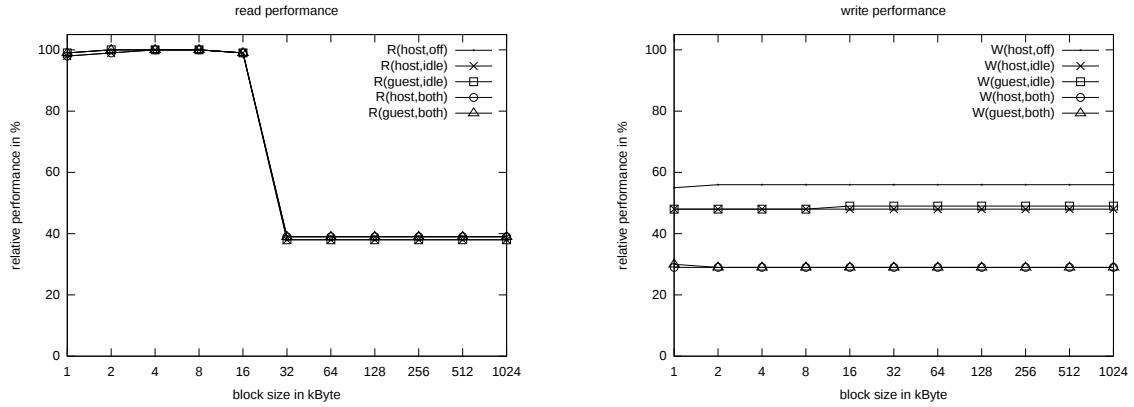


Figure 8.3.: Ramspeed benchmark results.

value is the mean value of $R(host, off)$ over all results with a block size smaller or equal than 16 kByte. This value is used as a reference for read and write performance measurement. For the interpretation of the results, it necessary to mention, that both machines use a 16 kB direct mapped cache with write-through strategy. The cache size can be easily figured out in the read performance chart. The write through strategy is the reason for the constant lines in the write performance chart. The general conclusion for the cache/memory access performance measurement is: Both, the host and the guest system, perform identical. Reduced performance for both systems results from the shared memory controller. The amount of performance reduction is the same for both.

8.4.2. Memory related Results

The proof of concept demonstrator uses a shared memory controller. It implements an GMMU vor virtual memory virtualization of the guest system as presented in section 4.2.2. In detail, it implements the contiguous chunk of physical memory solution utilizing a GMBR and GMLR.

As the proof of concept demonstrator is working, the usability of this virtual memory virtualization solution has been shown. The impacts of using a shared memory controller on the overall computational performance has already been discussed in the processor related section above.

8.4.3. Device Sharing related Results

The proof on concept demonstrator implements a physically support emulation interface for guest device virtualization (see page 38). In detail, the demonstrators uses this interface for providing hard disk functionality to the guest system, which

itself has no hard disk at all.

As the proof of concept demonstrator is working, the usability of this device virtualization approach has been shown.

On page 37, the possibility to physically share device interfaces between a guest and host machine is discussed. None of the proposed possibilities has been implemented by the proof of concept demonstrator, because the demonstrator has been developed to show the overall usability of the main idea. Implementing all possibilities of device sharing is not essential for this prove.

8.4.4. Machine related Results

FPGA Design Flow Limitations The proof of concept demonstrator implements a singleton guest system inside the reconfigurable area. It would be possible to instantiate a second guest system in parallel. However, the current partial reconfiguration design flow of Xilinx [Xil10b] only allows to change a partial reconfigurable logic area entirely. Hence, both system can be brought into the reconfigurable area only simultaneously. In consequence touching one system would therefore also affect the other one. A solution would be the introduction of another reconfigurable area, but instantiating another guest system there, is just the same as in the first reconfigurable area.

The "reconfigure a partial reconfigurable area only entirely" limitation is also the reason for another feature, proposed by the main idea of this thesis, but not implemented in the proof of concept demonstrator: adding additional devices to the guest machine, when it is already configured and the guest operating system is running.

The "reconfigure a partial reconfigurable area only entirely" limitation also applies to an entire FPGA. The introduction of several partial reconfigurable areas is no solution for the mentioned problem, as the interfaces of partial modules need to be fixed at synthesis time. A dynamic device instantiation for a runtime defined guest system bus, isn't applicable for a synthesis time defined interface.

An additional limiting factor of the current partial reconfiguration design flow is the necessity to pre-synthesize all configurations, that might be used by the virtualization platform. An *on demand* assembling of a machine is therefore a time consuming process, as the demanded machine has to be synthesized entirely "as required".

The Pause, Suspend and Resume problem. Another feature, not implemented in the proof of concept demonstrator, but supported by "conventional" virtualization platforms is the possibility to *pause, suspend and resume* and a virtual machine. To enable those features, it is necessary to save the state of a virtual machine entirely. For virtual machines, following the main idea of this thesis, the "state" of a guest

system is given by:

1. The contents of the guest systems main memory and the current state of all guest related processes/drivers in the host system. Both are summarized as the software state of a guest system.

2. The configuration of the partial reconfigurable area, the guest is instantiated in. It is given by the configuration stream, and forms the statical part of a systems hardware state.

3. The current state (value) of each storage element (Flip-Flop, Latches, Block RAMs, Main Memory etc.) associated with the guest. This is regarded as the dynamic part of a systems hardware state.

On a *pause* the dynamic part has to be "frozen". Additionally, on a *suspend*, the dynamic part needs to be stored somewhere, so that it can be bought back onto the static part and "unfrozen" on a *resume*. The proof of concept demonstrator doesn't provide the possibility to retrieve the dynamic part entirely. It's possible to get the contents of main memory. The values of the Flip-Flops or Latches of the configuration are not retrievable by the proof of concept demonstrator.

9. Conclusion

9.1. Summary

In this thesis, the idea of combining reconfigurable computing and virtualization to build hardware supported system virtual machines has been presented.

Several aspects of the idea have been discussed in theory. This included the question, how the overall virtualization system needs to be organized. Additionally, the interface requirements for a reconfigurable logic device, used to instantiate hardware, on which a guest operating can be executed have been investigated.

Main focus was direct on the requirements to adapt conventional virtualization methods on the different aspects of a machine. Therefore, the paradigm, that all resources of the overall virtualization system have to be managed by the host operating system, was a challenge. This included the question of how the guest systems virtual memory can be mapped onto the physically available address space. Also, the question, how access to I/O devices is provided for the guest systems has been discussed.

For testing purposes the Partial Reconfigurable Heterogeneous System (PRHS) framework has been implemented. This framework includes anything necessary to instantiate a virtualization system, based on the idea of combining reconfigurable computing and system virtual machines. This includes hardware, entirely self implemented in VHDL. The framework also includes an adapted Linux kernel (L4PRHS) and additional software to be used as operating system, running on this hardware, and a cross compiler toolchain.

The overall applicability of the idea has been proven by implementing a proof of concept demonstrator based on the PRHS framework. The demonstrator includes several key aspects of the presented theoretical discussion.

The benefits of a reconfigurable logic based virtualization system in relation to conventional virtual machine systems have also been discussed in theory and evaluated practically using the proof of concept demonstrator.

Those benefits are:

1. Lower computational performance loss for a guest system, caused by the virtualization mechanisms itself.

2. Native execution of a guest operating system, which is not ISA compatible with the host system, is possible. This results in computational performance gains compared to conventional system virtual machines, where different ISAs can only be virtualized by emulation or binary translation.

3. Stronger security and supervision policies can be enforced as guest and host separation is not only software, but also hardware based.

4. For type-1 hypervisors, no special requirements (Popek and Goldberg theorem) have to be fulfilled by the host processor.

Nevertheless, the proof of concept demonstrator shows two main problems for hardware supported virtualization systems:

1. The current demonstrator lacks the problem of being unable to pause, suspend and resume a guest system as this requires the ability to get and set the state of the guest system.

2. The current design flow for FPGAs or only partial reconfigurable areas results in a time consuming synthesis process, every time a new type of guest machine, is required. A straightforward *add an additional device to an already running guest machine* (by reconfiguration) is not possible, yet.

9.2. Future Prospects

The problems presented above need to be solved in the future. This will significantly enhance the applicability of the proposed idea to use reconfigurable logic to build hardware supported virtual machines.

9.2.1. Thoughts on the Pause, Suspend and Resume Problem

Due to the pending pause, suspend and resume problem, a guest systems blocks the reconfigurable logic area it is configured in, till the guest system is either hard-reseted by the host system or explicitly shut down.

Solving the pause, suspend and resume problem would allow to time-share a reconfigurable. This time-sharing has to be on coarse grained time-scale to be reasonable, as a reconfiguration takes time (several ms).

The obstacle of the pause, suspend and resume problem is the inability to retrieve and set the dynamic hardware state of the guest system. In the remainder of this section two ideas to overcome this problem are presented.

Scan Chain Techniques

Scan chain techniques are a well known approach in design testing. The main idea behind scan chains is to introduce special testing circuits into a design. These circuits allow to retrieve or manipulate the contents of all storing elements of a system. This is achieved by connecting all storing elements in one or more chains, the contents of the storing elements can be pushed through. This allows to bring a system into a defined starting state. The system can then be run for a while. After it is paused the scan chain can be used to retrieve the current system state. An introduction, including a broad related work section, to scan chain techniques is given in [NM96].

The ability of retrieving and setting the dynamic state of a system, provided by scan chain techniques, are well suited for solving the pause, suspend and resume problem of the current proof of concept demonstrator implementation of this thesis.

The main problem of this solution would be the circuitry overhead to implement the scan chain. This results in higher reconfigurable logic consumption for the guest machine. Furthermore, the scan chain(s) have to be included into the whole guest machine. This will require an extensive redesign of the PRHS framework hardware sources. Another disadvantage will be the length of the chains, resulting in very time consuming retrieve contents/ set contents processes.

ICAP Readback Capabilities

This idea only applies to Xilinx FPGAs (Virtex4 and later version) as it depends on features of the embedded ICAP device. Presented information is based on the version specific (Virtex4, Virtex5, Virtex6, Virtex7) configuration user guides of Xilinx.

The ICAP device of Xilinx FPGAs is used to perform an in-system dynamic and partial reconfiguration process. The ICAP device doesn't even allow to write configuration data, as is done with the proof of concept demonstrator. ICAP also allows to read back the configuration data. By additionally using a *Capture_VirtexN* device, also embedded in the Virtex FPGAs, the contents of Flip-Flops can be read back. Thereby, the current dynamic state of the FPGA can be retrieved.

Initial register (Flip-Flop) values are included in the configuration stream. By using the read back data of the Flip-Flop values to modify the configuration stream, a retrieved system's dynamic state can be brought back onto the FPGA or parts of it.

As it is FPGA vendor specific, the general applicability of this solution for the pause, suspend and resume problem is questionable. Additionally, the process of extracting the current state of Flip-Flops and modifying the configuration stream is FPGA depended, as this process is based on placement information.

9.2.2. Thoughts on FPGA Design Flows

The limitations of the current FPGA and PR design flows are summarized above. This section presents an idea, that is worth further investigations in the future to overcome the mentioned design flow limitations.

Current FPGA design flows map a system, written in a Hardware description language, directly onto the FPGA, resulting in the mentioned limitations. A common idea to overcome this problem is to introduce an abstraction layer between the system HDL description and the FPGA. This provides flexibility as abstractions generally simplifies and hides details. This approach allows to make the FPGA design flow more flexible, faster and device independent.

This idea has already been presented by different authors (e.g. [FP98] [HSE+00] [MK11]).

A dynamic guest system constitution (add and remove dedicated devices from/to a machine by reconfiguration) would be possible if the idea of introducing a FPGA abstraction layer is further developed and adopted for reconfigurable logic based system virtual machines.

9.3. Application Areas for Reconfigurable Logic based System VMs

In the final section of this thesis, application areas for the idea of combining reconfigurable logic and system virtual machines are briefly discussed. This shall also initiate further investigations on reconfigurable logic based system virtual machines.

IT Security and Forensic

A guest system of a reconfigurable logic based virtualization system can be seen as hardware sandbox. This strongly implies the use of the proposed idea of this thesis in areas, where security is of important interest. For enhanced security the host system needs to be strongly secure, as compromising the host system compromises all guest systems. On the other hand, the security constraints for a guest system can be relaxed. The impact of the available hardware based supervising possibilities of the host system onto the degree of relaxing the security policies of a guest need to be further investigated.

In the area of computer forensics, reconfigurable logic based system VMs can provide new possibilities. On reason are again the new hardware based supervision policies. Another reason is the possibility to monitor a compromised guest at runtime, with-

out the need to time-share a singleton processor as on conventional virtualization systems. This approach might be beneficial to analyze malware, that is able to detect, whether or not, it is running in a guest system of a conventional virtual machine.

Data Centers

The reconfigurable logic based virtualization system can provide several advantages for data centers:

Replacement of old systems Some services, implemented several decades ago on special hardware, need to be provided till today. Reconfigurable logic based system VMs will allow to replace the old dedicated and energy consuming hardware by instantiating the required hardware functionality in reconfigurable logic and operate the old system as a guest VM of reconfigurable logic based virtualization server.

Special Hardware Requirements Occasionally used computer architectures can be provided as a guest VM on demand, instead of stockpiling them physically.

Application specific hardware Instead of providing general purpose hardware for running software/services on it, hardware, specialized for the needs of the software/service can be instantiated in reconfigurable logic.

Personal Computers

The benefits for data centers also apply to personal computers.

Bibliography

[Adv96] Advanced RISC Machines Ltd (ARM). *ARM810 Data Sheet*, 1996.

[BDM02] L. Benini and G. De Micheli. Networks on chips: a new soc paradigm. *Computer*, 35(1):70–78, 2002.

[Bro] Martin C. Brown. Build a gcc-based cross compiler for linux.

[ByMK+06] Muli Ben-yehuda, Jon Mason, Orran Krieger, Jimi Xenidis, Leendert Van Doorn, Asit Mallick, and Elsie Wahlig. Utilizing iommus for virtualization in linux and xen. In *In Proceedings of the Linux Symposium*, 2006.

[CH02] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, June 2002.

[Cor13] Altera Corporation. *Altera's User-Customizable ARM-Based SoC*, 2013. www.altera.com/soc.

[cra04] Cray xd1 datasheet, 2004.

[CRKH05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartmann. *Linux Device Drivers*. O'Reilly, 2005.

[Dal99] Michael Dales. The proteus processor - a conventional cpu with reconfigurable functionality. In *FPL '99: Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications*, pages 431–437, London, UK, 1999. Springer-Verlag.

[FP98] William Fornaciari and Vincenzo Piuri. Virtual fpgas: Some steps behind the physical barriers. In Jos Rolim, editor, *Parallel and Distributed Processing*, volume 1388 of *Lecture Notes in Computer Science*, pages 7–12. Springer Berlin Heidelberg, 1998.

[FSF] Inc. Free Software Foundation. Gcc, the gnu compiler collection.

[Fur02] S. Furber. *ARM-Rechnerarchitekturen für System-on-Chip-Design*. mitp-Verlag, 2002.

[GG00] Pierre Guerrier and Alain Greiner. A generic architecture for on-chip

packet-switched interconnections. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '00, pages 250–256, New York, NY, USA, 2000. ACM.

[GHSB08] D. Gohringer, M. Hubner, V. Schatz, and J. Becker. Runtime adaptive multi-processor system-on-chip: Rampsoc. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –7, April 2008.

[GLD00] Steve Guccione, Delon Levi, and Daniel Downs. A reconfigurable content addressable memory. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, IPDPS '00, pages 882–889, London, UK, UK, 2000. Springer-Verlag.

[Gr13] Ingo Grbner. Including an ethernet controller into the prhs-framework, 2013.

[HB09] Rhett M. Hollander and Paul V. Bolotoff. RAMspeed, a cache and memory benchmarking tool.
http://alasir.com/software/ramspeed/, 2009.

[HEW05] Jan Haase, Frank Eschmann, and Klaus Waldschmidt. The self distributing virtual machine (sdvm) - making computer clusters heal themselves. In *Proceedings of the 23rd IASTED International Multi-Conference PARALLEL AND DISTRIBUTED COMPUTING AND NETWORKS*, pages 193–198, Innsbruck, Austria, February 2005.

[HH09] Chun-Hsian Huang and Pao-Ann Hsiung. Hardware resource virtualization for dynamically partially reconfigurable systems. *Embedded Systems Letters, IEEE*, 1(1):19 –23, may 2009.

[HK09] Daniel Hallmannseder and Bernd Klauer. Compilerunterstützung für die Dynamische Rekonfiguration eines Mikroprozessors. In *PII Workshop*, Hamburg, 2009. Technische Informatik, Helmut-Schmidt-Universität.

[HM04] G. Herrmann and D. Müller. *ASIC - Entwurf und Test*. Fachbuchverl. Leipzig im Carl-Hanser-Verlag, 2004.

[HP07] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2007.

[HSE+00] Yajun Ha, P. Schaumont, M. Engels, S. Vernalde, F. Potargent, L. Rijnders, and H. De Man. A hardware virtual machine for networked reconfiguration. In *Rapid System Prototyping, 2000. RSP 2000. Proceedings. 11th International Workshop on*, pages 194 –199, 2000.

[HW97] John R. Hauser and John Wawrzynek. Garp: A mips processor with

a reconfigurable coprocessor. In *Proceedings of the FCCM'97*, pages 12–21, 1997.

[HW08] Andreas Hofmann and Klaus Waldschmidt. Sdvm^r: A scalable firmware for fpga-based multi-core systems-on-chip. In *Proceedings of the 2008 IEEE Computer Society Annual Symposium on VLSI*, ISVLSI '08, pages 387–392, Washington, DC, USA, 2008. IEEE Computer Society.

[Kab12] Stefan Kabutke. Including an cf-card-controller in the prhs-framework, 2012.

[Kla13] Bernd Klauer. The convey hybrid-core architecture. In Wim Vanderbauwhede and Khaled Benkrid, editors, *High-Performance Computing Using FPGAs*, pages 431–451. Springer New York, 2013.

[Las13] David Lassig. Including a ps/2 keyboard into the prhs-framework, 2013.

[Loo13] Alexander Look. Including a ps/2 mouse into the prhs-framework, 2013.

[MK11] Dominik Meyer and Bernd Klauer. Multicore reconfiguration platform an alternative to rampsoc. *SIGARCH Computer Architecture News*, 39(4):102–103, 2011.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[NM96] Robert B. Norwood and Edward J. McCluskey. Synthesis-for-scan and scan chain ordering. In *VTS*, pages 87–92. IEEE Computer Society, 1996.

[NZ04] Adronis Niyonkuru and Hans Christoph Zeidler. Designing a runtime reconfigurable processor for general purpose applications. In *IPDPS*, 2004.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[PH09] David A. Patterson and John L. Hennessy. *Computer Organization and Design:: The Hardware/Software Interface*. Morgan Kaufmann, 4th edition, 2009.

[PP04] Christian Plessl and Marco Platzner. Virtualization of hardware - introduction and survey. In *Proc. 4rd Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA04)*, pages 63–69, Las Vegas, NV, USA, 2004. CSREA Press.

[PS06] Kostas Pagiamtzis and Ali Sheikholeslami. Content-addressable mem-

ory (CAM) circuits and architectures: A tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41(3):712–727, March 2006.

[Raz94] Rahul Razdan. *PRISC: programmable reduced instruction set computers.* PhD thesis, Harvard University, Cambridge, MA, USA, 1994.

[RNP+] Thomas D. Richardson, Chrysostomos Nicopoulos, Dongkook Park, Narayanan Vijaykrishnan, Yuan Xie, Chita R. Das, and Vijay Degalahal. A hybrid soc interconnect with dynamic tdma-based transactionless buses and on-chip networks. In *VLSI Design*, pages 657–664. IEEE Computer Society.

[Sca01] G. Scarbata. *Synthese und Analyse Digitaler Schaltungen.* Oldenbourg, 2001.

[Sch03] Martin Schoeberl. Jop: A java optimized processor. In *On the Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2003)*, volume 2889, pages 346–359. Springer, 2003.

[Sha49] Claude E. Shannon. The synthesis of two-terminal switching circuits. *Bell Systems Technical Journal*, 28:59–98, 1949.

[SKKR11] Martin Schoeberl, Stephan Korsholm, Tomas Kalibera, and Anders P. Ravn. A hardware abstraction layer in java. *ACM Trans. Embed. Comput. Syst.*, 10(4):42:1–42:40, 2011.

[SN05a] J.E. Smith and R. Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, 2005.

[SN05b] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[Tan07] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[TL10] Kuen Hung Tsoi and Wayne Luk. Axel: A heterogeneous cluster with fpgas and gpus. In *FPGA'10: Proceedings of the 18th annual ACM/SIGDA international on FPGAs*, pages 115–124, 2010.

[UHT] Craig Ulmer, Ryan Hilles, and David Thompson. Reconfigurable computing aspects of the cray xd1. Technical report, Sandia Nataional Laboratories.

[UKJCC12] Z. Ullah, M. Kumar Jaiswal, Y.C. Chan, and R. C C Cheung. Fpga implementation of sram-based ternary content addressable memory. In

Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International, pages 383–389, 2012.

[Vla] Denys Vlasenko. busybox.

[Wei84] Reinhold P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM*, 27(10):1013–1030, October 1984.

[Xil08] Xilinx Inc. *(DS080) System ACE Compact Flash Solution*, 2008.

[Xil10a] Xilinx Inc. *Memory Interface Solutions*, 2010. UG085.

[Xil10b] Xilinx, Inc. *Partial Reconfiguration User Guide*, 2010. http://www.xilinx.com.

[Xil10c] Xilinx Inc. *Virtex-6 Memory Interface Solutions*, 2010. UG406.

[Xil11] Xilinx Inc. *Data2MEM User Guide*, 2011. UG658.

[Xil12a] Xilinx Inc. *XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices*, 2012. UG687.

[Xil12b] Xilinx Inc. *Zynq-7000 All Programmable SoC Overview*, 2012. DS190.

A. VHDL Coding Conventions for PRHS Framework

Coding conventions are a well known aspect of software engineering. As the Hardware parts of the PRHS Framework should be easily reusable, it is necessary to define a naming convention and general coding rules:

A.1. Signal Naming Conventions

The names of internal signals, inputs and outputs all follow the same scheme as given in the following.

$$\{i|o|z|s|r|v\}\{d|c|e\}[n][\text{union name}]\text{descriptive signal name}$$

Figure A.1.: General structure of a signal name.

The meanings of the different components are explained in the following description:

i entity input

o entity output

z entity in- and output (tristate)

s internal signal

r internal register value

v internal variable

d data signal

c control signal (controls the flow of data signals)

e enable signal (special control signal, enabling register write or tristate output enable)

n (optional) indicates an active-low signal (active-high is regular)

union name for associated signals (e.g. different bus components) should have a union name, to easily figure out the association

descriptive signal name the real signal name

A.2. Component Instantiation Conventions

VHDL supports different types of component instantiation. The first and straightforward one is to skip the component declaration in the declarative part of an architecture and use `work.<component_name>` syntax to instantiate a component. This introduces a big problem for Xilinx synthesis flow: the entity, corresponding to the component, has to be compiled before the entity, the component is instantiated in.

Hence, for PRHS framework for each instantiated component, a corresponding component declaration has to be included in the declarative part of an architecture. Don't us the `work.<component_name>` syntax.

B. Devices - Addressing, Detection and Interrupts

The following table gives an overview on device present **line** (see *PRHS Bus Controller* device), **int**errupt line and devices for the platfrom supporting the proof of concept demonstrator of this work. The information is given on a device instance name basis; device class is given in brackets; base address for memory mapped I/O device registers are given as 32 bit hexadecimal values per device instance.

line	int.[1]	ML505 / XUPv5	ML605
0	x	ClockEventTimer (timer4prhs): $f0000020$ ClockSourceTimer (timer4prhs): $f0000030$	
1	x	uart0 (uart4prhs) : $f0001000$	
2	x	uart1 (uart4prhs) : $f0001010$	
3	x	reconfIF4prhs0 (reconfIF4prhs): control interface : $f1000000$ SD RAM interface : dynamically allocated	
4		SysAce4prhs0 (SysAce4prhs) : $d0000000$	
5	x	PS2_0 (pstwo4prhs) : $f0002000$	unused
6	x	PS2_1 (pstwo4prhs) : $f0002010$	unused
7	x	Ethernet_Ctrl0 (v5emac4prhs) : $f0010000$	unused
8-31		unused	