2 Virtualizable Architecture for embedded MPSoC

2.1 The Term Virtualization

Virtualization is a concept, which enables transparent resource sharing by strictly encapsulating modules. These virtualized modules behave as being the sole user of a resource. As each virtualized module has no knowledge about actually not being the sole user of a resource, malicious interference between virtualized modules is avoided by design. A survey lists the IBM VM Facility/370 to be the first machine realizing this concept [Goldberg 1974]. When the virtualized modules are software tasks, virtualization may be regarded as being a very strict interpretation of multi-tasking.

The use of virtualization is widespread among personal computers, servers, and mainframes. The isolated environment of virtualized operating systems, e. g., enable the simple cloning of systems or the backup and recovery of entire system states. For parallel architectures, approaches as the Parallel Virtual Machine exist, where virtualized tasks may share a pool of processors [Sunderam 1990]. This led to approaches as the SDVM [Haase 2004], which allows virtualized tasks to spread among a cluster of processors. However, in the field of embedded computing, especially regarding multi-processor architectures, the trend of exploiting virtualization is adopted with a certain delay. At least, in 2011, hardware-supported virtualization has been proposed for the ARM architecture, which is one of the most common processor architecture in today's embedded systems [Varanasi 2011]. In [Cohen 2010], processor virtualization accompanied by runtime compiling enables migrating software to heterogeneous processor platforms. This tool chain aims at the redistribution of software, but does not account for actual multi-processing.

Virtualization of software is usually accompanied by a virtualization host, i. e., a kernel or operating system, which manages the virtualized software. However, as embedded systems often face the constraints of harsh timing requirements as well as a limited amount of available memory, the overhead introduced by a virtualization solution may render its application as not feasible for most designs. In [Heiser 2008], this overhead is reduced by the introduction of microkernels, i. e., kernels with a very small memory footprint. As discussed before, in [Heiser 2008] two requirements are stated for embedded virtualization: A strong encapsulation of virtualized modules as well as a high communication bandwidth. The work of [Kopetz 2008] furthermore states principles for component composability in System-on-Chip (SoC) which are crucial in order to maintain the system's stability: interface specification, stability of prior services, non-interfering interactions, error containment, fault masking. In order to provide a virtualization scheme, these requirements and principles have to be considered. Thus, the strict encapsulation of virtualized modules and the transparency property of the virtualization procedure have to be maintained at all times.

Since virtualization is not exclusively limited to software being virtualized, several works regarding hardware module virtualization exist. As a first step, multi-tasking schemes known from the software world were adopted for hardware modules in [Brebner 1996]. In the succeeding work, the abilities of this task management are expanded by the partial reconfiguration feature of FPGAs [Brebner 2001]. The work of [Huang 2009] virtualizes hardware components in HW-SW designs. In doing so, several tasks can access virtualized instances of the same hardware component and, thus, reduce the waiting time arising from resource sharing. Task switching procedures for hardware tasks were proposed, e. g., in [Simmler 2000, Jozwik 2012, Stoettinger 2010], with the last one explicitly targeting virtualization features. Another approach targets the virtualization of whole FPGAs [Figuli 2011, Sidiropoulos 2013]. Here, so-called virtual FPGAs may be mapped to different underlying hardware. In doing so, a re-use of hardware blocks written for the virtual FPGA on different target devices is enabled.

Besides virtualization in live systems, virtual platforms may also be exploited for rapid system prototyping, test, and verification [Leupers 2012]. For virtual platforms, an accurate virtual image of an envisaged SoC is created, on which debugging and verification takes places. Accuracy may cover, e. g., execution time behavior and a power model of the underlying SoC.

Nevertheless, only few comprehensive approaches for virtualization in an embedded multi-processor environment exist. E. g., the concept of the SDVM was transferred to FPGAs by providing a dedicated firmware running on embedded processors in order to execute virtualized tasks [Hofmann 2008]. Another virtualization solution exploiting an MPSoC based on heterogeneous processor arrays was addressed in [Hansson 2011]. There, an underlying operating system acts as a host for virtualized software. However, this virtualization concept adds a fair amount of complexity to the system, hardening debug and verification. This architecture is exploited in [Ferger 2012] to host virtualizable, self-adaptable tiles. Splitting the computational power provided by the underlying MPSoC into tiles is one way to cope with the complexity of such architectures.

As the presented approaches often either rely on dedicated architectures or add a decent amount of complexity, which hardens design and testing, the following sections will, thus, introduce a very fast, simple, and memory-efficient virtualization concept for an array of embedded off-the-shelf processors, which will form a virtualizable MPSoC. As the virtualization properties are inserted by means of a hardware layer, the tasks may run natively on the processors without need for an underlying kernel.

2.2 Virtualization for Embedded Multi-Processor Architectures

Before motivating, why a virtualization procedure for an embedded multi-processor system is desirable, the characteristics of embedded processors are outlined in short. These characteristics as well as the desired properties lead to a set of requirements, which are defined in Section 2.2.3 and which have to be fulfilled by a virtualization procedure. Consequently, the steps necessary to meet these requirements are then outlined.

2.2.1 Characteristics of Embedded Processors

Despite the fact that we see ourselves surrounded by computer devices in our everyday's life, such as personal computers, notebooks, or tablet computers, the overwhelming number of computers is embedded.[1] In almost all powered devices, we may find digital circuits. The more functionality a device offers, the higher is the probability, that at least one embedded processor is exploited for this purpose. As embedded devices often have a very narrow scope of predefined uses, and, moreover, face several constraints, which will be addressed below, the characteristics of processors exploited in embedded devices differs from those employed in personal computers.

A personal computer serves a wide range of applications, e. g., writing documents, managing photos of the last vacation, listening to music, or playing video games. Therefore, a personal computer is designed to fulfill all these needs in an acceptable manner. Thus, the processor employed in a personal computer is equipped with a huge instruction set in order to speed up a broad range of applications.[2] This design concept is known as Complex Instruction Set Computer (CISC). In order to further boost execution speed by executing several applications in parallel, modern processors feature more than one processor core on the chip. A CISC design in combination with multi-core layout comes at price. Recent processors for personal computers are implemented on more than 2 billion transistors [Intel Corporation 2011]. Moreover, such processors are outlined for a Thermal Design Power of 77 W and above [Intel Corporation 2013a, p. 43]. For embedded designs, however, such characteristics are often unwanted.

Embedded computers are designed to handle a specific, reduced set of applications based on the intended use of the device. In contrast to variability and maximum performance, other aspects are of higher importance. As the range of applications is narrow and known during design of the device, a smaller instruction set for a so-called Reduced Instruction Set Computer (RISC) architecture may be exploited.[3] A

[1] According to the Community Research and Development Information Service (CORDIS) of the European Commission, embedded processors account for 98 % of all produced processors in 2006 [Research 2006].

[2] For reference, the "Intel 64 and IA-32 Architectures Software Developer's Manual" lists 434 different processor instructions [Intel Corporation 2013b].

[3] The manual for the Xilinx MicroBlaze RISC processor, which will be exploited for the prototype implementation in the scope of this work, lists 87 instructions [Xilinx, Inc. 2012] – a fraction of the 434 instructions provided by the Intel 64 and IA-32 CISC architectures.

simplified chip design furthermore reduces the transistor count. Consequently, the most significant constraint for embedded designs may be met: device cost. An embedded processor has to be as cheap as possible in order to lower the overall device cost. At the expense of this constraint, the performance of embedded processors is usually lower than that of those employed in personal computers. As a consequence of a lower transistor count and a simpler chip design, the power consumption is also lower. Since many of today's embedded computers, such as smartphones or mp3 players, are mobile, reducing energy consumption is an essential requirement to lengthen battery life. The simpler chip design is further accompanied by a fairly reduced interfacing. This will be an important property regarding the envisaged shift of task execution.

As low device cost is desired, embedded soft-core processors may be targeted. In contrast to usual, hard-wired integrated circuits, the so-called hard-cores, a soft-core processor is solely represented either by a behavior description given in a hardware description language (HDL), such as Verilog or VHDL or by a netlist. The description of a soft-core processor may be transformed into a hardware design by a synthesis process. Here, the description is mapped to primitives of the targeted chip, e. g., to look-up tables and registers of an FPGA.

Soft-core processors feature advantages, which are not present for hard-core processors. Due to their representation in an HDL, their behavior may be modified by altering their corresponding hardware description. However, the vendors of commercial soft-core processors may restrict the modification, e. g., by encrypting the files containing the hardware description. Nevertheless, open-source processors, such as the PicoBlaze [Xilinx, Inc. 2013b] or the Secretblaze [Barthe 2011], offer the modification of their hardware description.

Besides this manual adaption, soft-core processors may feature several pre-defined customizations. Based on the intended use, e. g., floating-point units, multipliers, or barrel shifters may be activated. During synthesis, the corresponding hardware descriptions of the desired functionality are included. Therefore, in contrast to a hard-wired processor, a soft-core processor may be tailored to the application purpose by disabling unnecessary functions and, thus, cost is reduced by a resulting lower resource consumption.

As a soft-core processor is available by its hardware description, multi-processor systems may easily be designed by instantiating the processor description multiple times in a top-level hardware description. Thus – given sufficient resources of the targeted FPGA – a multi-processor system may be designed without an increase in cost for the additional soft-core processors, whereas for a hard-core processor design, each additional core also causes additional cost. However, there are currently no common multi-core soft-core processors available yet. Therefore, multi-processing on soft-core processors relies on instantiating a set of soft-core processors.

Main drawback of soft-core processors is performance, which is usually lower than that of hard-wired embedded processors. A hard-wired processor, whose placement has underwent several optimization steps will always outperform a processor with

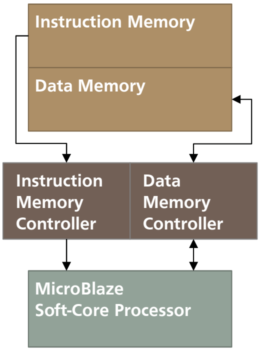


Figure 2.1: A MicroBlaze Processor System.

similar behavior, whose description has to be mapped onto existing chip primitives. For the first one, structures and routes on the chip may be tailored to the processor, whereas for the latter one, the processor is tailored to the target chip.

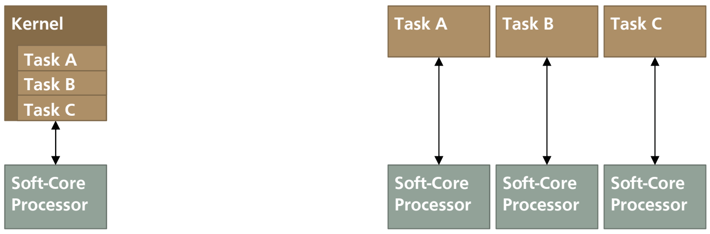
This work will demonstrate the virtualization concept by exploiting the commercial MicroBlaze soft-core processor, which is provided by the device vendor Xilinx, Inc. [Xilinx, Inc. 2013a]. The files containing the hardware description are encrypted and cannot be modified by a designer. However, since this work will demonstrate an approach to enable virtualization features without modifying existing processor designs, a modification of the hardware description is neither necessary nor desired.

The MicroBlaze is a common 32-bit RISC processor, which features a five stage pipeline in its default set-up. It is designed as Harvard architecture; therefore, instruction and data memory are separated from each other. After synthesis, instructions and data reside in BlockRAM (BRAM), memory primitives on FPGAs. Instructions and data are transferred between memory and the processor by two dedicated memory controllers. A processor system containing the MicroBlaze IP core, as well as instruction and data memories with their corresponding memory controllers is depicted in Figure 2.1. Further details of the processor architecture are provided later in this work as they get of particular importance.

Before highlighting the enhancements of the processor architecture depicted in Figure 2.1 towards a virtualizable multi-processor design, the intended purpose of the virtualization procedure is motivated.

2.2.2 Purpose of Virtualization in Embedded Multi-Processor Arrays

As highlighted above, virtualization enables transparent task handling on a host processor. While this functionality is well-established in the field of personal computing and server systems, embedded systems, especially soft-core processor-based designs, often lack of a virtualization property. To motivate why a virtualization concept



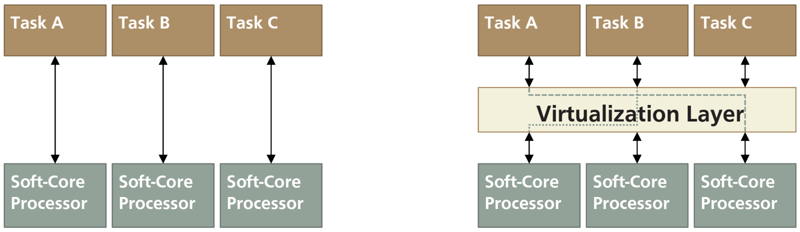
(a) Kernel. **(b)** Dedicated Processor Resources.

Figure 2.2: Task Management by a Kernel (a) compared to dedicated Processor Resources for each Task (b).

enhances embedded system design, two common design alternatives for embedded processor systems are discussed.

In the first alternative, a kernel manages the access of tasks to a processor, cf. Figure 2.2, left hand side. Both kernel and tasks reside in the same memory area. In this system, all the tasks and the kernel are statically bound to the processor. The employment of a kernel eases the scheduling of tasks on the processor. Furthermore, individual tasks may dynamically be excluded from processor access or may temporarily get a higher priority assigned. Unlike for personal computers, memory in embedded devices is often of limited size. Therefore, despite the convenient task handling, a kernel may add an unwanted overhead in terms of memory. Additionally, the switching process between tasks is time consuming. The Xilkernel, a kernel for the MicroBlaze processor provided by the device vendor Xilinx, Inc. takes approximately 1.360 clock cycles to switch between tasks. As embedded systems may face harsh timing constraints, the additional time required for a task switch is possibly not acceptable. In addition, if an embedded system is employed in a safety-critical environment, the usage of a kernel may pose a significant safety risk in case of address space violations of tasks if no memory management is exploited. All the tasks in the system reside in the same memory. A faulty task might thus alter memory sections of a task relevant to security. Therefore, in many systems that feature safety-critical tasks, switching tasks by a kernel is avoided and the second alternative as follows is chosen.

In the second alternative, each task features a dedicated processor, cf. Figure 2.2, right hand side. Since there are no other tasks running on a processor, there is no need for scheduling or for an underlying kernel. This eliminates the overhead both in terms of memory and time caused by a kernel. Moreover, aside from task communication, e.g., via buses, tasks are logically and physically separated from each other. This prevents harmful mutual task interference. Furthermore, since each task may occupy all of the processor time of its processor, performance may be higher compared to a single processor system that features a kernel. Drawback of this solution is the tremendous resource overhead.



(a) Static Bindings. **(b)** Virtualization Layer.

Figure 2.3: Resolving static Task-to-Processor Bindings (a) by Introduction of a Virtualization Layer between Tasks and Processors (b).

As an alleged solution, a hybrid solution could be exploited. Here, each safety-critical task might be bound to a dedicated processor, while other tasks in the system might share processor access via a kernel. However, the two design alternatives as well as this hybrid solution face the same major drawbacks. Due to the static bindings, it is not possible to either compensate for a defective processor at runtime or to adapt the task-processor bindings depending on current computing requirements. None of these solutions exploits the benefits arising from multi-processing, such as dynamically binding tasks to processors or executing a task in parallel on several processor instances.

As a consequence, a desired system has to feature the benefits of both designs. Tasks may share a processor resource despite avoiding the overhead caused by a kernel. In addition, strict task encapsulation has to be ensured to prevent task interference. Moreover, the solution has to exploit multi-processing, i. e., tasks may be dynamically allocated to and scheduled on a set of processors. In order to compensate for faulty devices, mechanisms for fault detection and the ability to switch from a faulty to a functioning processor have to be provided without the need to allocate a set of resource-wasting spare processors.

Virtualization as a mean of transparent task handling may enable all these features. This is achieved by the introduction of a so-called *Virtualization Layer* between tasks and processors. Otherwise isolated single-processor systems are thereby transformed into a flexible multi-processor system, cf. Figure 2.3. The static binding between tasks and processors is replaced by the ability to dynamically define new task-to-processor bindings.

As detailed in the previous section, embedded processor designs often face other constraints than processor systems for personal computers. Thus, in the following section, out of the characteristics of embedded processors and the intended purpose of a virtualization processor for an embedded processor array, a set of requirements is derived.

2.2.3 Requirements for Embedded Virtualization

For personal computing or server systems, cheap memory is available in huge numbers. Hard-disk drives, RAM, several levels of on-chip cache, and, last but not least, a set of registers provide a hierarchy of memory. In embedded systems, where cost reduction is crucial, memory is scaled as small as possible. Therefore, enabling virtualization features may not lead to a significant increase in memory needed for the multi-processor system.

Additionally, embedded systems may face timing constraints. Thus, a virtualization concept may not add a significant timing overhead and may not delay task execution. Consequently, if several tasks share a processor resource, the interruption of a running task in order to activate a task that has to fulfill its timing requirement has to be supported at any point in time.

Furthermore, embedded systems may be employed in safety-critical environments, such as in autopilots for plane navigation or, as outlined in Chapter 4.1, as driver assisting systems in an automotive environment. Correct system behavior is required since faulty computations may lead to severe incidents. This leads to two considerations. First, in an embedded system in a safety-critical environment, a task may not be harmfully affected by any other task in the system. Typical solutions completely, i. e., physically and logically, separate safety-critical tasks from other tasks in the system. Thus, mutual task interference has also to be strictly avoided for a system with virtualization features. Second, intrinsic mechanisms to detect or even mask faults are desired.

If multiple tasks are involved in an embedded system, the question about scheduling these tasks on the processor resources arises. A typical solution for scheduling issues is the exploitation of an embedded operating system or, with less overhead in terms of resources and time, a kernel. For the targeted MicroBlaze, several kernel types are available, such as the Xilkernel or Linux kernels. These kernels reside in the same memory as tasks and manage task scheduling. However, as discussed above, the overhead both in time and memory resources by a kernel is often unwanted. Thus, the usage of an existing kernel or operating system has to be avoided.

Enabling virtualization features for a set of processors will require enhancements to existing single-core processor designs. However, a solution that relies on the modification of a given processor architecture is limited to this specific processor type. Moreover, not all soft-core processors, despite being deployed as a textual hardware description, are modifiable, such as the MicroBlaze. Therefore, the virtualization features have to be as processor-independent as possible without dependence to processor-specific properties. At least, a possible migration to other processor types has to be supported with reasonable effort.

As for the processors, also the software of the tasks should not be undergo the need of a modification. Since most of embedded code is legacy code that is reused, a solution that required the re-writing of entire software projects is not feasible.

Given all these considerations, the following requirements towards an embedded virtualizable multi-processor system may be postulated:

1. Fast and transparent switch of task execution
2. No significant memory overhead caused by virtualization features
3. Guaranteed activation and interruption of task execution at any point in time
4. Strict encapsulation and isolation of tasks
5. Mechanisms for fault detection and masking
6. Avoidance of a common operating system or kernel residing in task memory
7. No modification of the processor core, usage of off-the-shelf processors
8. Minor or no modification of existing software code of tasks

In order to fulfill these requirements, the following sections will present the prerequisites necessary to enable virtualization features.

2.2.4 Prerequisites to enable Virtualization Features

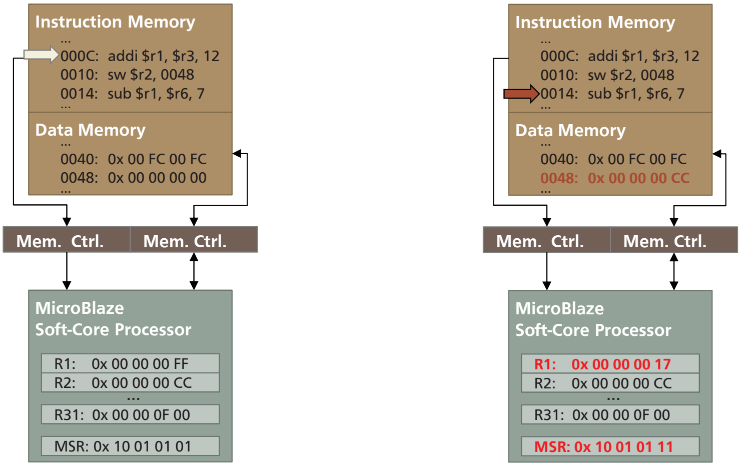
In order to enable virtualization features under consideration of the requirements postulated above, some prerequisites have to be taken. As the virtualization concept is built around the transparent switch of tasks, it has to be defined at first, which information of a task has to be considered and preserved during a task switch. Second, the enhancements to existing default soft-core processors systems as depicted in Figure 2.1, which will enable virtualization properties, are detailed.

Consideration of Task Context

As the virtualization procedure is intended to interrupt and resume task execution at any point in time, the current state of a task has to be saved during its deactivation. The current state of a task is defined by its *context*.

A context typically includes the instruction memory, the current content of the data memory, the current program counter address of the processor pointing to the next instruction to be fetched, and, last but not least, the internal state of the processor, i. e., the content of its registers, during task execution. Figure 2.4 illustrates the context of a task running on a MicroBlaze processor for the two points in time t_1 and t_2. Changes in the context that arise during execution of the task are highlighted in red for t_2. The arrow indicates the current program counter address.

Given the context information at a certain point in time, the current state of a task is exactly defined. In a common multi-tasking operating system, the switch between tasks is handled in software. Here, e. g., the contents of the processor's registers, which hold data of the task to be deactivated, are written in a stack memory.



(a) Point in Time t_1. **(b)** Point in Time t_2.

Figure 2.4: Dynamic Context of a Task over Time.

Upon this task's reactivation, the content is read back into the processor's register set. Accordingly, the proposed virtualization solution aims at extracting a task's context during the deactivation phase and at restoring this context during its reactivation. A comparable approach for an embedded multi-processor system was highlighted in [Beaumont 2012]. However, only a subset of the processors' registers was considered. In contrast, the virtualization procedure will take the full context inside a processor core into consideration. For hardware modules with internal states, the works of [Kalte 2005] and [Levinson 2000] highlight context extraction, which is based on a readback of the FPGA's configuration. While a readback of the FPGA configuration could also be exploited to determine the current state of a software task, this would limit the presented approach to FPGA architectures and, furthermore, would require an off-chip resource managing this context extraction. Thus, the present approach will handle task context extraction in a more convenient way independent of the actual chip architecture. The context elements to be saved during a virtualization procedure are now discussed in short.

Instruction and Data Memory The instruction memory is a read-only memory and, therefore, its state does not change during task execution. Thus, no special treatment is needed for the instruction memory. During task execution, the data memory is read and written, cf. Figure 2.4. With each write operation, the state of the data memory is altered. During deactivation of a task, the current state of the data memory has to be preserved and prevented from being altered. This may easily be achieved, e. g., by detaching the data memory from the processor during the deactivation phase.