Adaptively Reconfigurable Controller for the Flash Memory

Ming Liu[1,2], Zhonghai Lu[2], Wolfgang Kuehn[1] and Axel Jantsch[2]
[1]*Justus-Liebig-University Giessen*
[2]*Royal Institute of Technology*
[1]*Germany*
[2]*Sweden*

1. Introduction

As the continuous development on the capacity and work frequency, Programmable Logic Devices (PLD) especially Field-Programmable Gate Arrays (FPGA) are playing an increasingly important role in embedded systems designs. The FPGA market has hit about 3 and 4 billion US dollars respectively in 2009 and 2010, and is expected by Xilinx CEO Moshe Gavrielov to grow steadily to 4.5 billion by the end of 2012 and 6 billion by the end of 2015. The application fields of FPGAs and other PLDs range from bulky industrial and military facilities to portable computer devices or communication terminals. Figure 1 demonstrates the market statistics of some most significant fields in the third quarter of 2009.

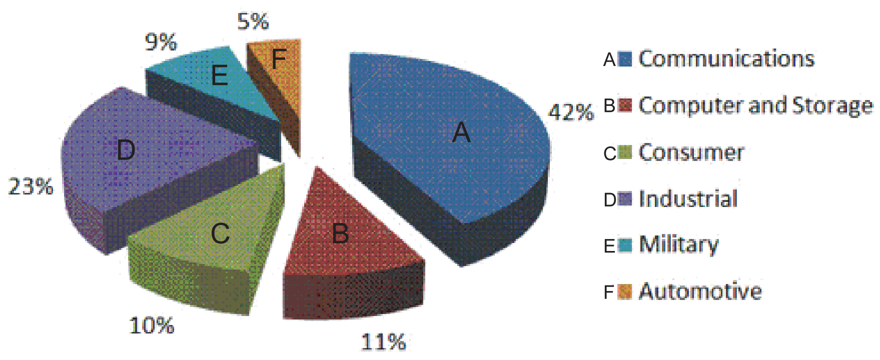


Fig. 1. PLD market by end applications in the third quarter of 2009 (Dillien, 2009)

FPGAs were originally used as programmable glue logic in the early period after its birth. Due to the capacity and clock frequency constraints at that time, they typically worked to bridge Application-Specific Integrated Circuit (ASIC) chips by adapting signal formats or conducting simple logic calculation. However at present, modern FPGAs have obtained enormous capacity and many advanced computation/communication features from the semiconductor process development; they can accommodate complete computer systems consisting of hardcore or softcore microprocessors, memory controllers, customized hardware accelerators,

as well as peripherals, etc. Taking advantage of design IP cores and interconnection architecture, it has become a reality to easily implement System-on-Programmable-Chip (SoPC) or system-on-an-FPGA.
In spite of large advances, the chip area utilization efficiency as well as the clock speed of FPGAs is still very low in comparison with ASICs. One of the reasons is that FPGA employs Look-Up Table (LUT) to construct combinational logic, rather than primary gates as in ASICs. In (Kuon & Rose, 2006), the authors have measured FPGAs to be 35X larger in area and 3X slower in speed than a standard cell ASIC flow, both using 90-nm technology; In (Lu et al., 2008), a 12 year old *Pentium*® design was ported on a Xilinx Virtex-4 FPGA. A 3X slower system speed (25 MHz vs. 75 MHz) is still observed, although the FPGA uses a recent 90-nm technology while the original ASICs were 600-nm. The speed and area utilization gap between FPGAs and ASICs has been additionally quantified in (Zuchowski et al., 2002) and (Wilton et al., 2005) for various designs. Therefore we understand that FPGA programmable resources are still comparatively expensive. Efficient resource management and utilization remain to be a challenge especially for those applications with simultaneous high performance and low cost requirements.
Flash memory is often used to store nonvolatile data in embedded systems. Due to its intrinsic access mode, normally it does not feature as high speed read and write operations as volatile memories such as Dynamic Random Access Memory (DRAM) or Static Random Access Memory (SRAM). In many applications, flash memory is only used to hold data or programs which are expected to be retrievable after each time power off. It is only addressed very occasionally or even never during the system run-time, when those data or programs have already been loaded in the main memory of the system. For example, an embedded Operating System (OS) kernel may be loaded from the flash into DDR for fast execution in case of system power-on. Afterwards, the flash memory will never be addressed in systems operation unless the OS kernel is scheduled to be updated. Because of the occasionality of flash accesses, it generates resource utilization inefficiency if the flash memory controller is statically mapped on the FPGA design but does not operate frequently.
In the recent years, an advanced FPGA technology called Dynamic Partial Reconfiguration (DPR or PR) has emerged and become gradually mature for practical designs. It offers the capability to dynamically change part of the design without disturbing the remaining system. Based on the FPGA PR technology, which enables more efficient run-time resource management, we present a peripheral controller reconfigurable system design in this chapter: A NOR flash memory controller can be multiplexed with other peripheral components (in the case study an SRAM controller), time-sharing the same hardware resources with all the required system functionalities realized. We will elaborate the design in the following sections.

2. Conventional static design on FPGAs

2.1 Static design approach

A peripheral controller is the design component which interfaces to the peripheral device and interprets or responds to access instructions from the CPU or other master devices. So a flash memory controller is the design by which CPU addresses external flash chips. Figure 2 shows the top-level block diagram of a flash memory controller for the Processor Local Bus

(PLB) (IBM, 2007) connection. It receives control commands from the PLB to read from and write to external memory devices. The controller design provides basic read/write control signals, as well as the ability to configure the access time for read, write, and recovery time when switching between read and write operations. In addition, the memory data width and the bus data width are parameterizable. They can be automatically matched by performing multiple memory cycles when the memory data width is less than PLB. This design structure is capable of realizing both synchronous and asynchronous device access. It may also support other parallel memory accesses with small modification effort, such as SRAM.

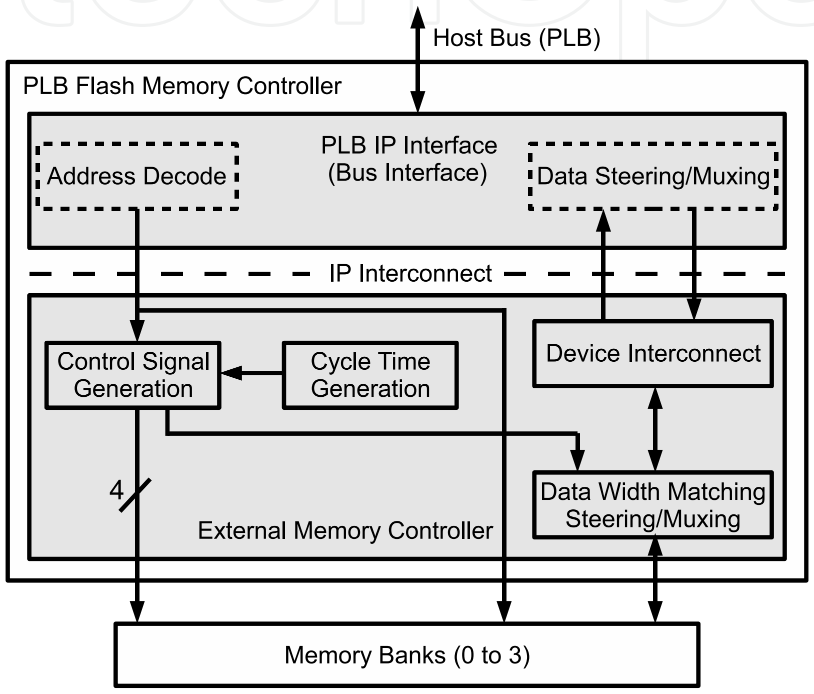

Host Bus (PLB)
PLB Flash Memory Controller
PLB IP Interface (Bus Interface)
Address Decode
Data Steering/Muxing
IP Interconnect
Control Signal Generation
Cycle Time Generation
Device Interconnect
4
External Memory Controller
Data Width Matching Steering/Muxing
Memory Banks (0 to 3)

Fig. 2. Top-level block diagram of the PLB flash memory controller (Xilinx, 2006)

Figure 3 demonstrates a typical system-on-an-FPGA design for embedded applications. As an example, we adopt the Xilinx Virtex-4 FX FPGA for the implementation. We observe that all components are interconnected by the PLB, including the microprocessor, memory controllers, the application-specific algorithm accelerator as well as peripheral devices. In case of system power-on, the FPGA firmware bitstream is firstly downloaded to configure the FPGA via a special configuration interface (Dunlap & Fischaber, 2010). Afterwards an embedded Linux OS kernel is loaded by a bootloader program into the main memory of DDR for fast execution. In the design, a NOR flash memory stores nonvolatile data necessary for system startup in the field, including both the bitstream file and the OS kernel.

Suppose we are constructing a system aiming at memory bandwidth hungry computation for certain applications. Hence a Zero-Bus Turnaround (ZBT) SRAM is integrated in the system in addition to the main DDR memory. The SRAM is utilized as a Look-Up Table (LUT) component by the algorithm accelerator to carry out application-specific computation. It features higher data bandwidth and more efficient data movement than DDR. With the

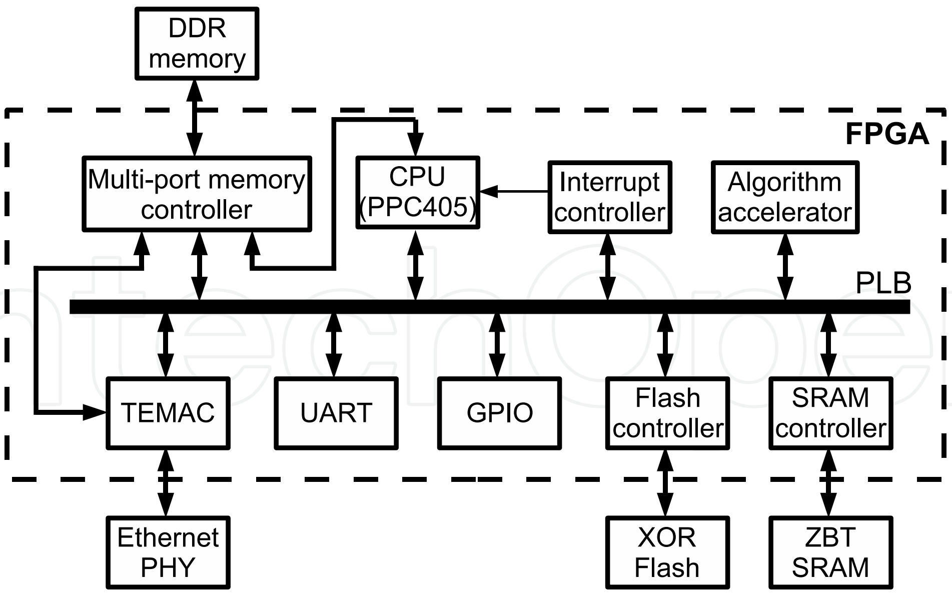


Fig. 3. Static design on an FPGA. The system is bus-based and all components are connected to the PLB. We may see that both the flash controller and the SRAM controller are concurrently placed in the design with the conventional static approach.

conventional static design approach, both the flash and the SRAM controller are concurrently placed on the FPGA in order to address the two types of memories.

2.2 Motivation

The flash memory is used to hold nonvolatile data for in-field system startup. It will be rarely addressed during the system operation unless external management commands require the bitstream or the OS kernel to be updated. On the other hand, application-specific computation starts only after the FPGA firmware is configured and the OS is successfully booted. Therefore on account of the occasionality of flash access as well as the operation exclusiveness between flash and SRAM, it generates resource utilization inefficiency if the flash controller is permanently mapped on the FPGA design but does not function frequently. Hence we consider to make the flash memory controller dynamically changeable and time-share the same on-chip resources with the SRAM controller.

3. FPGA partial reconfiguration technology

Modern FPGAs (e.g. Xilinx Virtex-4, 5, and 6, Altera Stratix 5 FPGAs) offer the partial reconfiguration capability to dynamically change part of the design without disturbing the remaining system. This feature enables alternate utilization of on-FPGA programmable resources, therefore resulting in large benefits such as more efficient resource utilization and less static power dissipation (Kao, 2005). Figure 4 illustrates a reconfigurable design example on Xilinx FPGAs: In the design process, one Partially Reconfigurable Region (PRR) A is reserved in the overall design layout mapped on the FPGA. On the early-stage dynamically reconfigurable FPGAs (e.g. Xilinx Virtex-II and Virtex-II Pro), PRR reservation must run through a complete slice column, because a slice column is the smallest load unit of a configuration bitstream frame (Hubner et al., 2006; Xilinx, 2004). With respect to the latest FPGA generations (e.g. Xilinx Virtex-4, 5, and 6), PRRs can be the combination of slice squares. Various functional Partially Reconfigurable Modules (PRM) are individually implemented within the PR region in the implementation process, and their respective partial bitstreams are generated and collectively initialized in a design database residing in a memory device in the system. During the system run-time, various bitstreams can be dynamically loaded into the FPGA configuration memory by its controller named Internal Configuration Access Port (ICAP). With a new module bitstream overwriting the original one in the FPGA configuration memory, the PRR is loaded with the new module and the circuit functions according to its concrete design. In the dynamic reconfiguration process, the PRR has to stop working for a short time (reconfiguration overhead) until the new module is completely loaded. The static portion of the system will not be interfered at all.

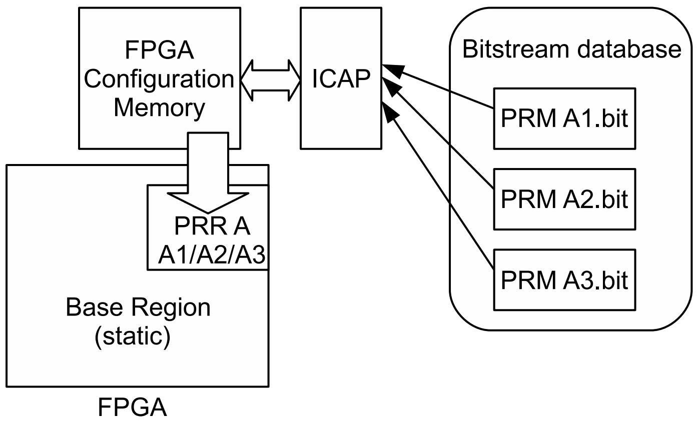


Fig. 4. Partially reconfigurable design on Xilinx FPGAs

The ICAP primitive is the hardwired FPGA logic by which the bitstream can be downloaded into the configuration memory. As shown in Figure 5, ICAP interfaces to the configuration memory and provides parallel access ports to the circuit design based on programmable resources. During the system run-time, a master device (typically an embedded microprocessor or Direct Memory Access (DMA)) may transfer partial reconfiguration bitstreams from the storage device to ICAP to accomplish dynamic reconfiguration. The complete ICAP design, in which the ICAP primitive is instantiated, interfaces to the system interconnection fabric to communicate with the processor and memories. In (Liu, Kuehn, Lu & Jantsch, 2009), (Delorme et al., 2009) and (Liu, Pittman & Forin, 2009), the authors explore the design space of ICAP IP module and present optimized designs. Through using either DDR or SRAM memories to hold partial bitstreams, these designs may achieve a run-time reconfiguration throughput of about 235 MB/s or close to 400 MB/s. The reconfiguration time overhead is linearly proportional to the size of partial bitstreams. Thus, a typical modular design of several tens or hundreds of KiloBytes in the partial bitstream requires several tens up to hundreds of microseconds (μs) for run-time reconfiguration.
The PR technology is coupled very closely to the underlying framework of the FPGA chip itself. We use the Xilinx FPGAs to explain the PR design flow as illustrated in Figure 6: The design begins from partitioning the system between the static base design and the reconfigurable part. Usually basic hardware infrastructures that expect continuous work

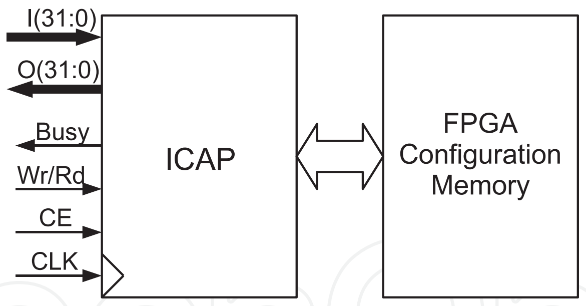


Fig. 5. The ICAP primitive on Xilinx FPGAs

and do not want to be unloaded or replaced during the operation are classfied into the static category, such as the system processor or the main memory controller. The partially reconfigurable part delegates those modules with dynamically swapping needs in the PR region. All the modular designs including PRMs are assembled to form an entire system. After synthesis, netlist files are generated for all the modules as well as the top-level system. The netlists serve as input files to the FPGA implementation. Before implementation, the Area Group (AG) constraints must be defined to prevent the logics in PRMs from being merged with the ones in the base design. Each PRR will be only restricted in the area defined by the RANGE constraints. Then after the following independent implementation of the base design and PR modules, the final step in the design flow is to merge them and create both the complete bitstream (with default PR modules equipped) and partial bitstreams for respective PR modules. Hence, the run-time reconfiguration process is initiated when one partial bitstream is loaded into the FPGA configuration memory and overwrites the corresponding segment.

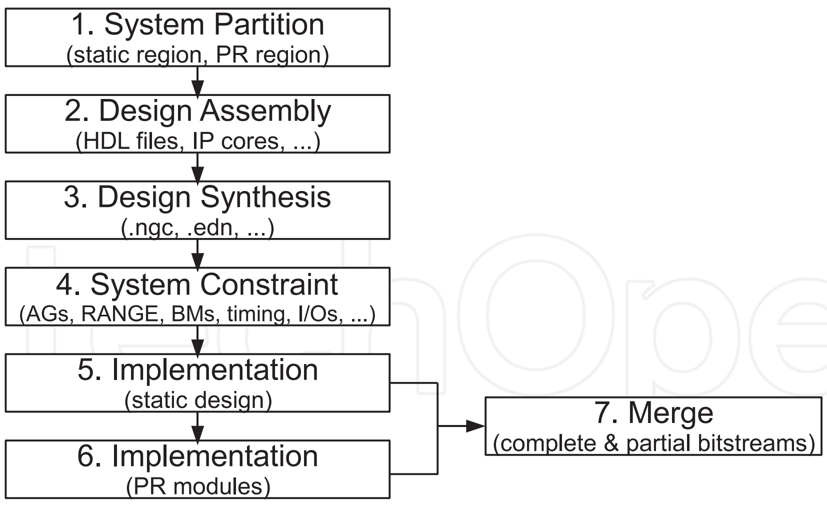


Fig. 6. Xilinx PR design flow

4. Design framework of adaptively reconfigurable peripherals

The modular design concept popularly adopted in static systems applies also to run-time reconfigurable designs on FPGAs. As we discussed in the previous section, the entire system is partitioned and different tasks are individually implemented as functional modules in dynamically reconfigurable designs. Analogous to software processes running on top of OSes and competing for the CPU time, each functional module can be regarded as a *hardware process* which is to be loaded into reconfigurable slots (i.e. PRRs) on the FPGA rather than General-Purpose microprocessors (GPCPU). Multiple hardware processes share the programmable resources and are scheduled to work according to certain types of disciplines on the awareness of computation requirements. Context switching happens when the current hardware process in charge of one task is leaving the reconfigurable slot (being overwritten) and another new task is to be loaded to start working. All these key issues in the adaptive computing framework are classified into and addressed within certain layers in hardware or software. Figure 7 demonstrates the layered hardware/software architecture and details in different aspects will be presented in the following subsections respectively.

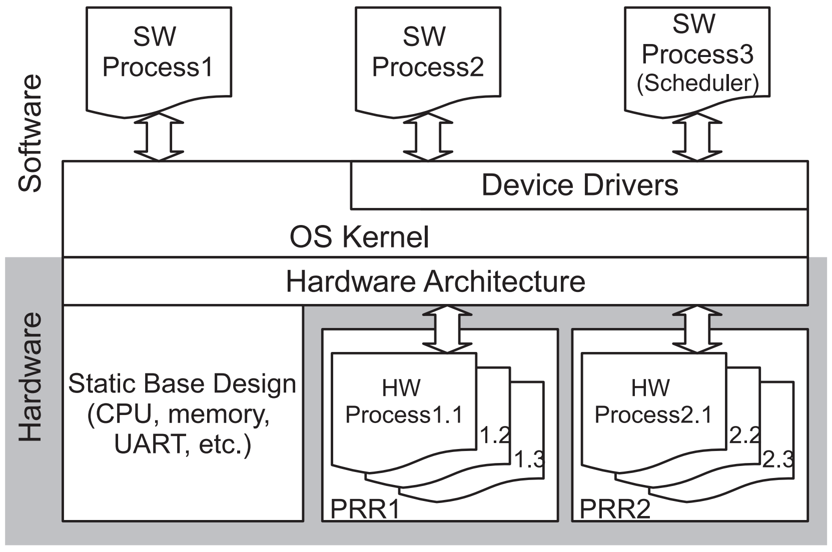


Fig. 7. Hardware/software layers of the adaptive reconfigurable system

4.1 Hardware structure

A dynamically reconfigurable platform may contain a general-purpose host computer system and application-specific functional modules. Figure 8 shows a system on a Xilinx Virtex-4 FPGA. Existing commercial IP cores can be exploited to quickly construct the general computer design, consisting of the processor core, the main DDR memory controller, peripherals, and the interconnection infrastructure using the PLB bus. In addition to the fundamental host computer system, run-time reconfigurable slots are reserved for being dynamically equipped with different functional modules. In the figure we show only one PRR to explain the principle. When incorporated in the PRR, PR modules communicate with the static base design, specifically the PLB bus for receiving controls from the processor and I/O buffers to external devices. Noting that the output signals of a PR module may unpredictably toggle during active reconfiguration, “disconnect” logic (illustrated in the callout frame in Figure 8) is required to be inserted to disable PRM outputs and isolate the unsteady signal state for the base design from being interfered. Furthermore, a dedicated “reset” signal aims to solely reset the newly loaded module after each partial reconfiguration. Both the “disconnect” and the separate "reset" signal can be driven by software-accessible General-Purpose I/Os (GPIO).

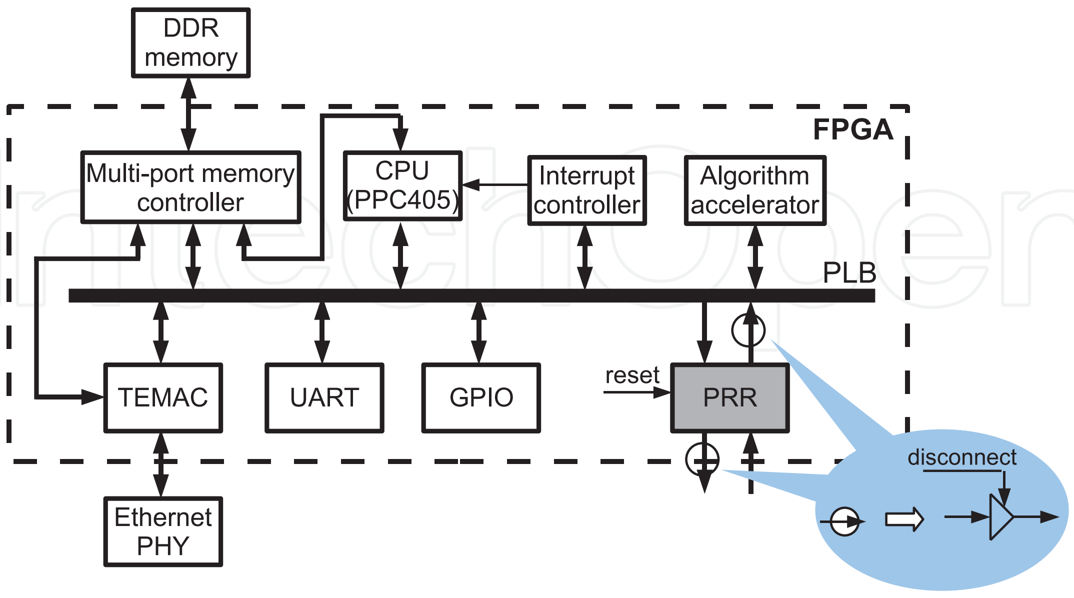


Fig. 8. The hardware infrastructure of the PR system

In the previous Xilinx Partial Reconfiguration Early Access design flow (Xilinx, 2008), a special type of component called Bus Macro (BM) must be instantiated to straddle the PR region and the static design, in order to lock the implementation routing between them. This is the particular treatment on the communication channels between the static and the dynamically reconfigurable regions. The BM components have been removed in the new PR design flow (Xilinx, 2010). They are no longer needed and the partition I/Os are automatically managed by the development software tool.
One significant advantage of this hardware structure, is that it conforms to the modular design appraoch: Different functional tasks are respectively implemented into IP cores. They are wrapped by the PLB interface and integrated in the bus-based system design. Normal static designs can be easily converted into a PR system by attentively treating the connection interface and mapping various functional modules in the same time-shared PR region. Little special consideration is needed to construct a PR system on the basis of conventional static designs.

4.2 OS and device drivers

As in conventional static designs, all hardware modules sharing a same reconfigurable slot can be managed by the host processor with or without OS support. In a standalone mode without OS, the processor addresses device components with low-level register accesses in application programs. While in OSes, device drivers are expected to be customized. In a Unix-like OS, common file operations are programmed to access devices (Corbet et al., 2005), including "*open*", "*close*", "*read*", "*write*", "*ioctl*", etc. Interrupt handlers should also be implemented if the hardware provides interrupt services.
Different device components multiplexed in a same PR region are allowed to share the same physical address space for system bus addressing, due to their operation exclusiveness on the

time axis. In order to match software operations with the equipped hardware component, two approaches can be adopted: Either a universal driver is customized for all the reconfigurable modules sharing a same PR region. Respective device operations are regulated and collected in the code. The ID number of PR modules is kept track of and passed to the driver, branching to the correct instructions according to the currently activated hardware module; or the drivers are separately compiled into software modules for different hardware components. The old driver is to be removed and the new one inserted, along with the presence of a newly loaded hardware device. Among these two approaches, the former one can avoid the driver module removing/inserting time overhead in the OS, while the latter one is more convenient for system upgrades when a new task is added to share a PR region.
Little special consideration or modification effort is required on the OS and device drivers for run-time reconfigurable systems in comparison with static designs. The most important thing to note, is to keep track of the presently activated module in the PRR and correctly match the driver software with the hardware. Otherwise the device module may suffer from misoperations.

4.3 Reconfiguration management

In dynamically reconfigurable designs, run-time module loading/unloading is managed by a scheduler. Analogous to the scheduler in an OS kernel which determines the active process for CPU execution, the scheduler in FPGA reconfigurable designs monitors trigger events and decides which functional module is to be configured next to utilize the reconfigurable slot. All hardware processes are preemptable and they must comply with the management from the scheduler. The scheduling policy may be implemented in hardware with Finit State Machines (FSM). However for more design convenience, it can be ported in the software application program running on the host processor with or without OS support. Distinguished from the kernel space scheduling in (So et al., 2006) and the management unit design in hardware in (Ito et al., 2006), the user space software scheduling possesses significant advantages of convenient portability to other platforms, avoidance of error-prone OS kernel modification, and flexibility to optimize scheduling disciplines. Scheduling policies are very flexible. But they have direct effect on the system performance and should be optimized according to concrete application requireiments, such as throughput or reaction latency. One general rule is to minimize the hardware context switching times, taking into account the dynamic reconfiguration time overhead and extra power dissipation needed during the reconfiguration process.
The scheduler program is only in charge of light-weight control work and usually does not feature intensive computation. In addition, the host CPU only initiates run-time reconfiguration by providing the bitstream storage address as well as the length, and it is actually the master block or the DMA component in the ICAP designs that transports the configuration data (Delorme et al., 2009; Liu, Kuehn, Lu & Jantsch, 2009; Liu, Pittman & Forin, 2009). Therefore dynamic reconfiguration scheduling does not typically take much CPU time, especially when the trigger events of module switching happen only infrequently and the scheduler is informed by CPU interrupts.

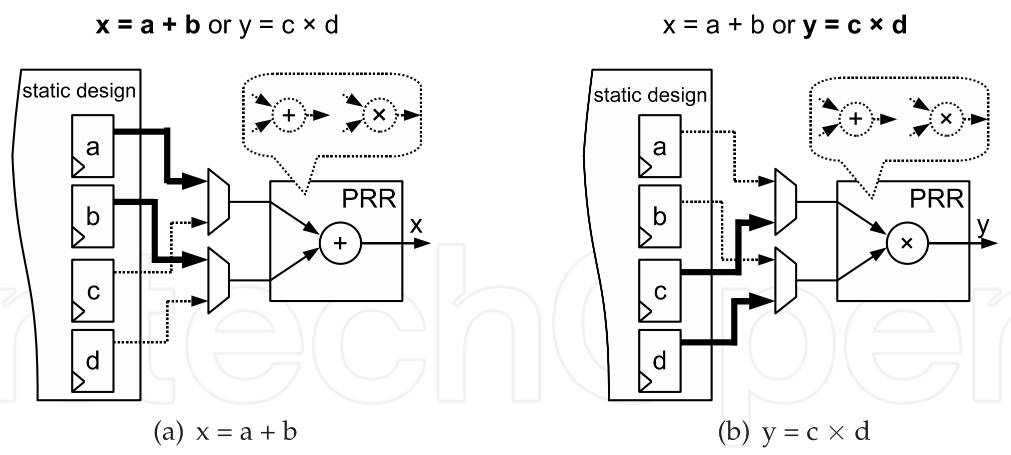


Fig. 9. Contextless module switching in the reconfigurable design

4.4 Context switching

The context of hardware processes refers to the buffered incoming raw data, intermediate calculation results and control parameters in registers or on-chip memory blocks residing in the shared resources of PR regions or static interface blocks. In some applications, it becomes contextless when the buffered raw data are completely consumed and no intermediate state is needed to be recorded. Thus the scheduler may simply swap out an active PR module. After some time when it resumes, a module reset will be adequate to restore its operation. Otherwise, context saving and restoring must be accomplished. Figure 9 and 10 respectively demonstrate these two circumstances: In the design in Figure 9, two dynamically reconfigurable functional modules (adder and multiplier) do not share the interface registers and they both feature pure conbinational logic in using the PRR. Hence during each time when the PRR is reconfigured with an arithmetic operator, the register values in the interface block are not needed to be saved or restored in order to obtain correct results of x and y. By contrast in the design of Figure 10, the operand registers in the static interface are shared and the reconfigurable region also contains the context of one operand for the addition operation. Therefore in case of module switching, the operands of the former operation must be saved in the system memory, and the ones for the recently resumed operator are to be restored.

Generally speaking, two approaches can be employed to address the context saving and restoring issue: In case of small amounts of parameters or intermediate results, register accesses can efficiently read out the context into external memories and restore it when the corresponding hardware module resumes (Huang & Hsiung, 2008). When there are large quantities of data buffered in on-chip memory blocks, the ICAP interface can be utilized to read out the bitstream and extract the storage values for context saving (Kalte & Porrmann, 2005). In order to avoid the design effort and large time overhead in the latter case, an alternative solution is to intentionally generate some periodic "pause" states without any context for the data processing module. Context switching can be then delayed by the scheduler until meeting a pause state.

4.5 Inter-process communication

Reconfigurable modules (hardware processes) placed at run-time may need to exchange data among each other. With respect to those modules that are located in different PR regions,

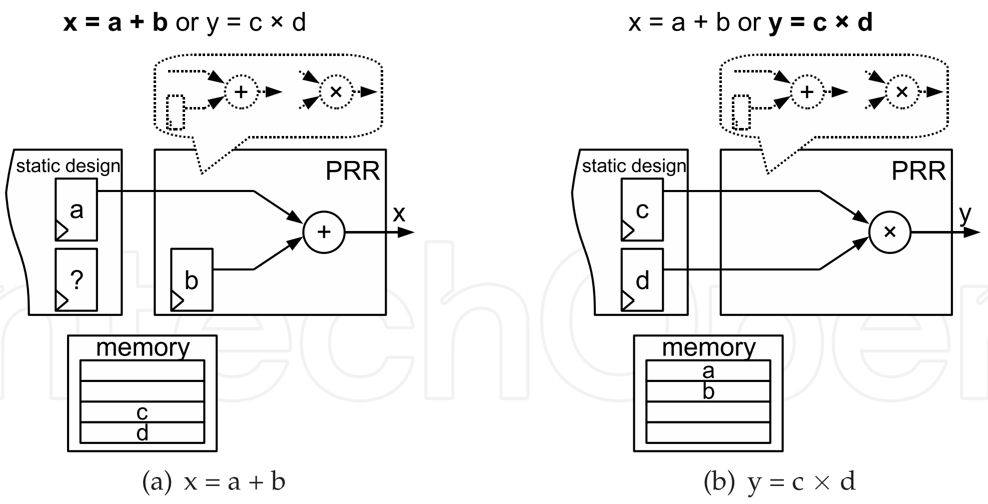


Fig. 10. Context saving and restoring in the reconfigurable design

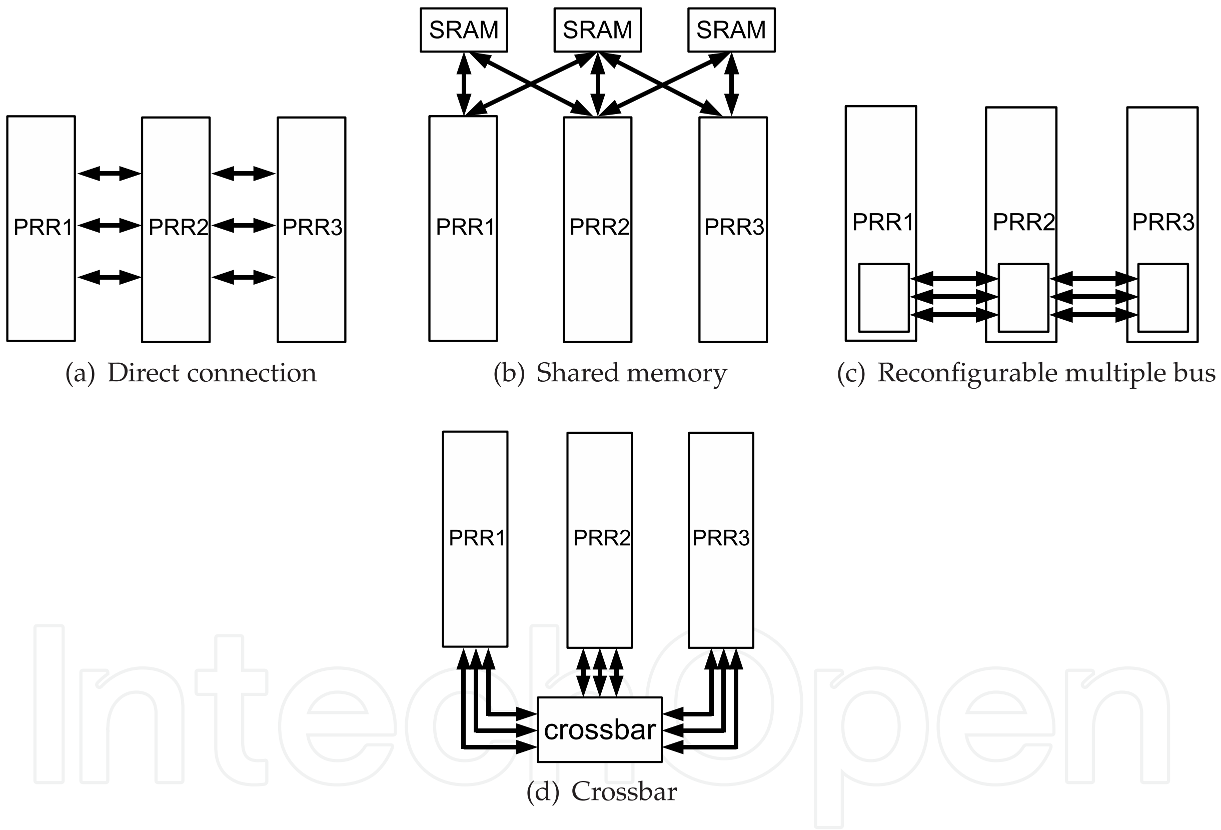
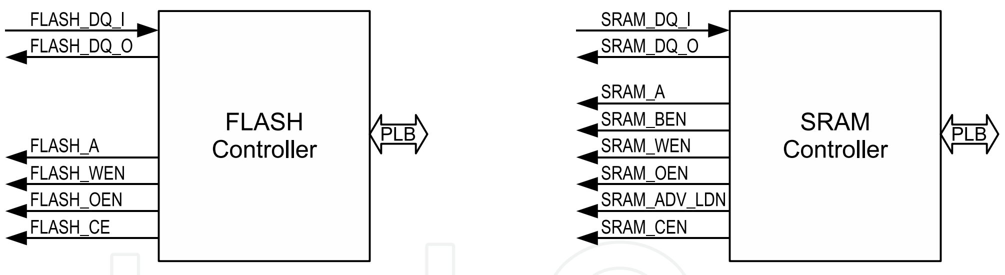


Fig. 11. Inter-process communication approaches among PRRs (Majer et al., 2007)

they can communicate through canonical approaches as in static designs. For example Figure 11 demonstrates some general solutions for inter-PRR communications, including direct connection, shared memory, Reconfigurable Multiple Bus (RMB) (Ahmadinia et al., 2005; Elgindy et al., 1996), and crossbar. Detailed description on these approaches can be found in (Majer et al., 2007) and (Fekete et al., 2006), in which inter-module communications have been intensively investigated in dynamically reconfigurable designs.



(a) Flash controller (b) SRAM controller

Fig. 12. Blackboxes of the flash controller and the SRAM controller

More generally, communications among PR modules that are time-multiplexed in the same reconfigurable slot may also exist and be required in the hardware implementation. In this circumstance, static buffer devices must be employed to hold the IPC information while the PRR is being dynamically reconfigured (Liu et al., 2010). As the producer module is active to work, the IPC information is injected into the buffer. Afterwards the consumer module may take the place of the producer in the reconfigurable slot and digest the IPC data destined to it. The buffer device can either directly interface to the PRR, or be located in the main memory and accessed via interconnection architectures such as the system bus. The concrete IPC using this approach for the reconfigurable controller design will be revisited in the next section.

5. Reconfigurable design of flash/SRAM controllers

5.1 Hardware/software design

Both the flash and the SRAM controllers are picked up from the Xilinx IP library. We do not concern their in-depth design details, but simply regard them as blackboxes instead with communication interfaces demonstrated in Figure 12. In the figure, the left side is the interface to external memory devices (Flash or SRAM) and the right side is to the system bus.

Figure 13 shows the hardware structure: An off-chip asynchronous NOR flash memory and a synchronous SRAM share the same data, address and control bus I/O pads of the FPGA. These two chips are exclusively selected by the “CE” signal. The flash and the SRAM controllers are both slave devices on the system bus. They are selectively activated in the reserved PRR by run-time partial reconfiguration. In order to isolate the unsteady output signals from the PRR during active reconfiguration, “disconnect” logic is inserted in both interfaces between the controllers and the PLB bus or external devices. Moreover, a dedicated “reset” signal takes charge of solely reseting the newly loaded module after each run-time reconfiguration. Both the “disconnect" and the separate “reset" signals are driven by a GPIO core under the control of the host processor.

An open-source Linux kernel runs on the host PowerPC 405 processor. To manage run-time operations in Linux, device drivers for hardware IP cores have been brought up to provide programming interfaces to application programs. We configure the open-source Memory Technology Device (MTD) driver (Woodhouse, 2005) to support NOR flash accesses in Linux. Other drivers are customized specifically for the LUT block in SRAM, PLB_GPIO and MST_HWICAP. With drivers loaded, device nodes will show up in the “/dev” directory of the Linux file system, and can be accessed by predefined file operations. The drivers are

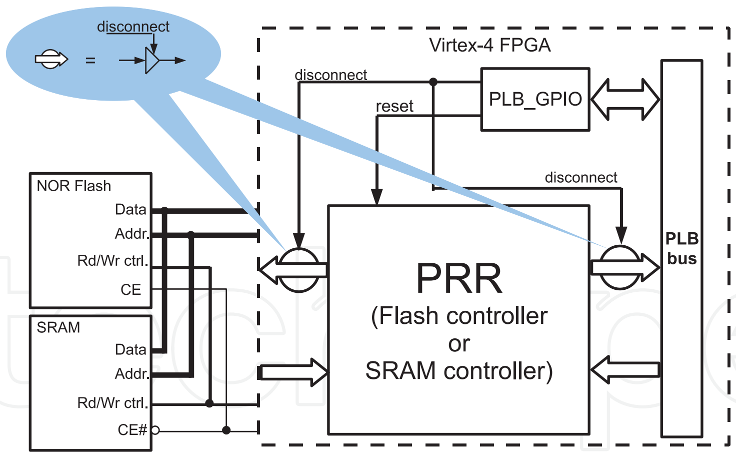


Fig. 13. Hardware structure of the flash/SRAM PR design

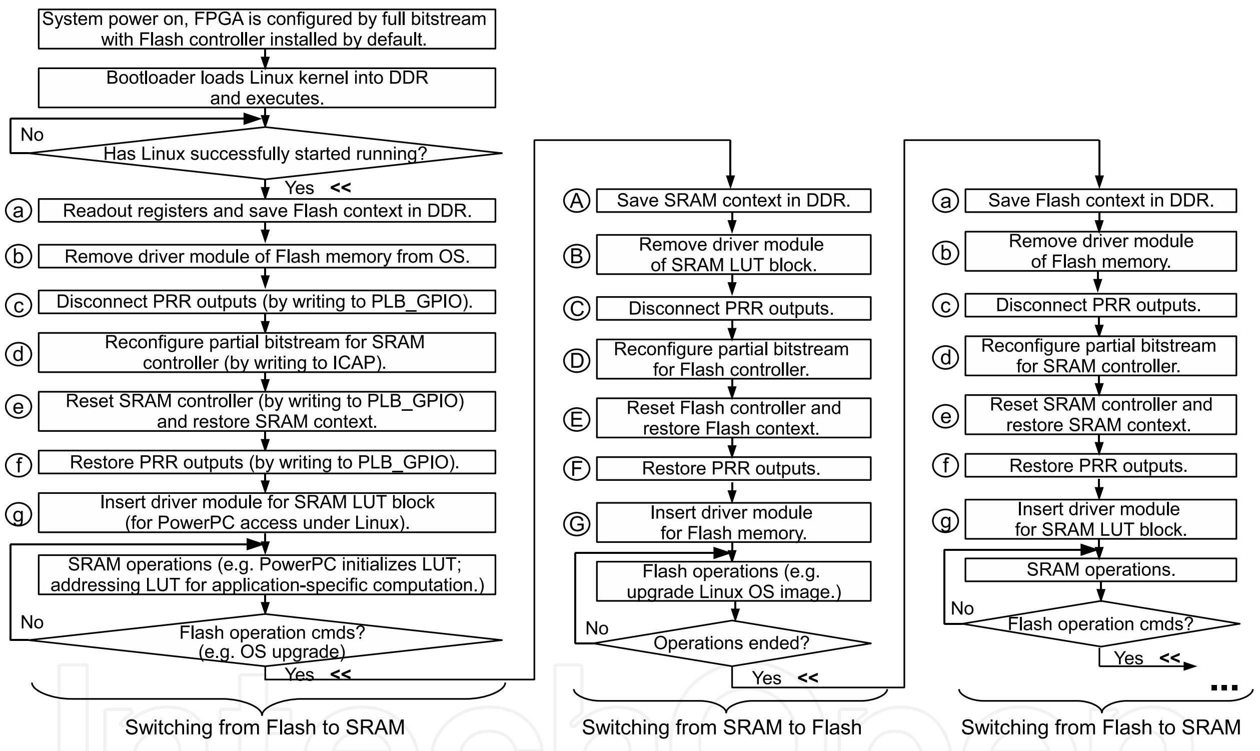

System power on, FPGA is configured by full bitstream with Flash controller installed by default.
Bootloader loads Linux kernel into DDR and executes.
No
Has Linux successfully started running?
Yes <<
a Readout registers and save Flash context in DDR.
b Remove driver module of Flash memory from OS.
c Disconnect PRR outputs (by writing to PLB_GPIO).
d Reconfigure partial bitstream for SRAM controller (by writing to ICAP).
e Reset SRAM controller (by writing to PLB_GPIO) and restore SRAM context.
f Restore PRR outputs (by writing to PLB_GPIO).
g Insert driver module for SRAM LUT block (for PowerPC access under Linux).
SRAM operations (e.g. PowerPC initializes LUT; addressing LUT for application-specific computation.)
No
Flash operation cmds? (e.g. OS upgrade)
Yes <<
A Save SRAM context in DDR.
B Remove driver module of SRAM LUT block.
C Disconnect PRR outputs.
D Reconfigure partial bitstream for Flash controller.
E Reset Flash controller and restore Flash context.
F Restore PRR outputs.
G Insert driver module for Flash memory.
Flash operations (e.g. upgrade Linux OS image.)
No
Operations ended?
Yes <<
a Save Flash context in DDR.
b Remove driver module of Flash memory.
c Disconnect PRR outputs.
d Reconfigure partial bitstream for SRAM controller.
e Reset SRAM controller and restore SRAM context.
f Restore PRR outputs.
g Insert driver module for SRAM LUT block.
SRAM operations.
No
Flash operation cmds?
Yes <<
...
Switching from Flash to SRAM
Switching from SRAM to Flash
Switching from Flash to SRAM

Fig. 14. Flow chart of multiplexing flash/SRAM in Linux

compiled into modules. They will be inserted into the OS kernel when the corresponding device hardware is configured, or removed when not needed any longer.

The hardware process scheduler is implemented in a C program. It detects the memory access requirements on flash or SRAM from either the system interior or external user commands, and meanwhile manages the work sequence of both types of memories. Figure 14 shows a flow chart, in which the scheduler alternately loads the flash and the SRAM controller with context awareness. During the device module reconfiguration, the Linux OS as well as the remaining hardware system keeps running without breaks. In this figure, steps labeled with "a - g" are used to dynamically configure the SRAM controller, and the ones labeled with

"A - G" are to load the flash controller. Events marked by the symbol "$\ll$" are detected by the scheduler to trigger hardware context switching. Main switching steps before device operations include:

1. To save the register context of the to-be-unloaded device in DDR variables if necessary.
2. To remove the driver module of the to-be-unloaded device from the OS.
3. To disconnect the PRR outputs for isolating its unsteady state during active reconfiguration from the static design.
4. To dynamically load the partial bitstream of the expected controller by initiating the MST_HWICAP core.
5. To solely reset the newly loaded device controller, and recover its register context if there exists.
6. To re-enable the PRR outputs, restoring the communication links from the PRR to the static design.
7. To insert the corresponding device driver in the OS, for the processor access with high-level application software.

After these steps, the recently equipped controller module becomes ready for memory accesses on the NOR flash or the SRAM.
In this design, IPC operations can be realized through the third-party shared memory such as DDR. For example when the system is just powered on, the SRAM LUT initialization data are retrieved from the nonvolatile flash and buffered in the system DDR memory. After the flash controller is unloaded and the SRAM controller is activated in the PRR by dynamic reconfiguration, the LUT data are then migrated into the SRAM chip for application-specific computation. The IPC data flow is illustrated in Figure 15.

5.2 Results

Through enabling either the flash controller or the SRAM controller with system self-awareness, multitasking has been accomplished within a single reconfigurable slot on the FPGA. Figure 16 demonstrates the rectangular shape of the reserved PR region on a Virtex-4 FX20 FPGA layout, as well as two controller implementations after place-and-route. The reconfigurable design results in a more efficient utilization of hardware resources, as listed in Table 1. We understand that both the flash memory controller and the SRAM controller must be concurrently placed in the static system design, implying a total resource consumption equivalent to the sum of both device modules. A PR region is reserved in the reconfigurable design, sufficiently large to accommodate all kinds of needed resources of both device modules. Moreover, a little more resource margin is added for the place-and-route convenience of the software tool. In contrast to the conventional static approach, we observe that the reconfigurable system saves 43.7% LUTs, 33.8% slice registers and 47.9% I/O pads, with both flash and SRAM services realized. The reduced resource requirement not only enables to fit a large system design on small FPGA chips for lower hardware cost, but also makes the I/O pads shared and simplifys the Printed Circuit Board (PCB) routing.

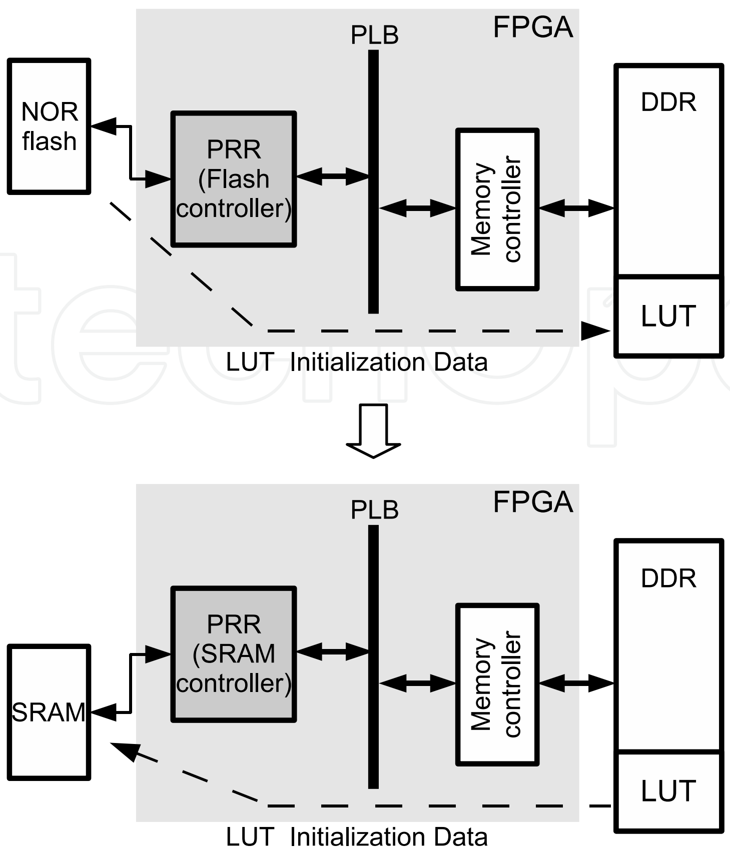
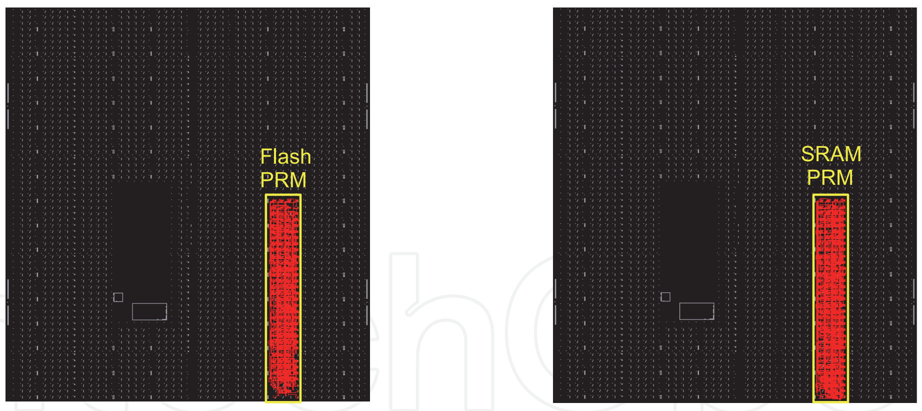


Fig. 15. Migrating LUT initialization data from the flash memory to the SRAM



(a) The flash implementation in PRR

(b) The SRAM implementation in PRR

Fig. 16. Implementation of the flash and the SRAM controller within the PRR on a Virtex-4 FX20 FPGA

6. Conclusion

Based on the FPGA run-time reconfigurability, we present a dynamically reconfigurable NOR flash controller for embedded designs. This technique is motivated by the operation occasionality of the flash memory and the resultant programmable resource waste on the FPGA, when adopting the conventional static development approach. We discuss the

Resources	Static flash controller	Static SRAM controller	Total	PRR	Resource saving
4-input LUTs	923	954	1877	1056	43.7%
Slice Flip-Flops	867	728	1595	1056	33.8%
I/O pads	56	61	117	61	47.9%

Table 1. Resource utilization of the static/reconfigurable flash/SRAM designs

design framework of adaptively reconfigurable peripherals in this chapter, concerning various aspects in hardware and software. In the practical experiment, a reserved reconfigurable slot is time-shared by the flash memory controller and an SRAM controller. Both system requirements of accessing the flash memory and the SRAM are equally accomplished in the reconfigurable design, with much less resource utilization of FPGA LUTs, slice registers as well as I/O pads.
This design technique is not limited to memory controller modules, but can apply to all kinds of modular devices operating exclusively. In addition, system functionalities can be later extended by adding more functional modules to time-share a same reconfigurable slot. It not only enhances the resource utilization efficiency on FPGAs, but also enables the possibility of future firmware upgrade without hardware modification.

7. Acknowledgment

This work was supported in part by BMBF under contract Nos. 06GI9107I and 06GI9108I, FZ-Jülich under contract No. COSY-099 41821475, HIC for FAIR, and WTZ: CHN 06/20. The authors also thank Xilinx Inc. for the software donation.

8. References

Ahmadinia, A., Bobda, C., Ding, J., Majer, M., Teich, J., Fekete, S. & van der Veen, J. (2005). A practical approach for circuit routing on dynamic reconfigurable devices, *Proceedings of the IEEE International Workshop on Rapid System Prototyping*, pp. 84–90.

Corbet, J., Rubini, A. & Kroah-Hartman, G. (2005). *Linux Device Drivers (Third Edition)*, O'REILLY & Associates, Inc.

Delorme, J., Nafkha, A., Leray, P. & Moy, C. (2009). New opbhwicap interface for realtime partial reconfiguration of fpga, *Proceedings of the International Conference on Reconfigurable Computing and FPGAs*, pp. 386–391.

Dillien, P. (2009). An overview of fpga market dynamics. SOCcentral webpage. URL: *http://www.soccentral.com*

Dunlap, C. & Fischaber, T. (2010). Partial reconfiguration user guide. UG702, Xilinx Inc.

Elgindy, H. A., Somani, A. K., Schroeder, H., Schmeck, H. & Spray, A. (1996). Rmb Íc a reconfigurable multiple bus network, *Proceedings of the International Symposium on High-Performance Computer Architecture*, pp. 108–117.

Fekete, S., van der Veen, J., Majer, M. & Teich, J. (2006). Minimizing communication cost for reconfigurable slot modules, *Proceedings of the International Conference on Field Programmable Logic and Applications*, pp. 1–6.

Huang, C. & Hsiung, P. (2008). Software-controlled dynamically swappable hardware design in partially reconfigurable systems, *EURASIP Journal on Embedded Systems* 2008: 1–11.

Hubner, M., Schuck, C. & Becker, J. (2006). Elementary block based 2-dimensional dynamic and partial reconfiguration for virtex-ii fpgas, *Proceedings of the International Parallel and Distributed Processing Symposium*.

IBM (2007). 128-bit processor local bus architecture specifications. Version 4.7, IBM Inc.

Ito, T., Mishou, K., Okuyama, Y. & Kuroda, K. (2006). A hardware resource management system for adaptive computing on dynamically reconfigurable devices, *Proceedings of the Japan-China Joint Workshop on Frontier of Computer Science and Technology*, pp. 196–202.

Kalte, H. & Porrmann, M. (2005). Context saving and restoring for multitasking in reconfigurable systems, *Proceedings of the International Conference on Field Programmable Logic and Applications*, pp. 223–228.

Kao, C. (2005). Benefits of partial reconfiguration, *Xcell Journal* Fourth Quarter: 65–67.

Kuon, I. & Rose, J. (2006). Measuring the gap between fpgas and asics, *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, ACM Press, pp. 21–30.

Liu, M., Kuehn, W., Lu, Z. & Jantsch, A. (2009). Run-time partial reconfiguration speed investigation and architectural design space exploration, *Proceedings of the International Conference on Field Programmable Logic and Applications*, pp. 498–502.

Liu, M., Lu, Z., Kuehn, W. & Jantsch, A. (2010). Inter-process communications using pipes in fpga-based adaptive computing, *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, p. 80.

Liu, S., Pittman, R. N. & Forin, A. (2009). Minimizing partial reconfiguration overhead with fully streaming dma engines and intelligent icap controller, *Technical Report MSR-TR-2009-150*, Microsoft Research.

Lu, S., Yiannacouras, P., Suh, T., Kassa, R. & Konow, M. (2008). A desktop computer with a reconfigurable *pentium*®, *ACM Transactions on Reconfigurable Technology and Systems* 1(1): 1–15.

Majer, M., Teich, J., Ahmadinia, A. & Bobda, C. (2007). The erlangen slot machine: A dynamically reconfigurable fpga-based computer, *The Journal of VLSI Signal Processing* 47(1): 15–31.

So, H. K., Tkachenko, A. & Brodersen, R. (2006). A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph, *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pp. 259–264.

Wilton, S., Kafafi, N., Wu, J., Bozman, K., Aken'Ova, V. & Saleh, R. (2005). Design considerations for soft embedded programmable logic cores, *IEEE Journal of Solid-State Circuits* 40(2): 485–497.

Woodhouse, D. (2005). Memory technology device (mtd) subsystem for linux. MTD webpage. URL: *http://linux-mtd.infradead.org/archive/index.html*

Xilinx (2004). Two flows for partial reconfiguration: Module based or difference based. XAPP290, Xilinx Inc.

Xilinx (2006). Plb external memory controller (plb emc) (2.00a). DS418, Xilinx Inc.

Xilinx (2008). Early access partial reconfiguration user guide for ise 9.2.04i. UG208, Xilinx Inc.

Xilinx (2010). Partial reconfiguration user guide. UG702, Xilinx Inc.

Zuchowski, P., Reynolds, C., Grupp, R., Davis, S., Cremen, B. & Troxel, B. (2002). A hybrid asic and fpga architecture, *Proceedings of the International Conference on Computer-Aided Design*, pp. 187–194.