# Partial Reconfiguration on FPGAs in Practice
—
# Tools and Applications

Dirk Koch[1], Jim Torresen[1], Christian Beckhoff[2], Daniel Ziener[3], Christopher Dennl[3], Volker Breuer[3], Jürgen Teich[3], Michael Feilen[4], and Walter Stechele[4]

[1] University of Oslo, Norway, Email: {dirk, jimtoer}@ifi.uio.no
[2] ReCoBus, Email: christian@recobus.de
[3] University of Erlangen, Germany,
Email: {daniel.ziener, teich}@informatik.uni-erlangen.de
[4] TU Munich, Germany, Email: {michael.feilen, walter.stechele}@tum.de

**Abstract.** Run-time reconfiguration of FPGAs has been around in academia for more than two decades but it is still applied very seldom in industrial applications. This has two main reasons: a lack of killer applications that substantially benefit from run-time reconfiguration and design tools that permit to quickly implement corresponding reconfigurable systems. This tutorial gives a survey on state-of-the-art trends on reconfigurable architectures and devices, application specific requirements, and design techniques and tools that are essential for implementing partial run-time reconfiguration on FPGAs. This is followed by a demonstration of the floorplanning and constraint generation tool GoAhead. Furthermore, the tutorial will reveal several applications that benefit from partial reconfiguration, including network data processing, digital signal processing, cognitive radio, and systems on a reconfigurable chip. For these applications, the individual challenges and implementation issues are presented together with the achieved results. This tutorial demonstrates that partial FPGA reconfiguration is beneficial and applicable in industrial systems.

## 1 Introduction

In the early days of field programmable gate arrays (FPGAs), the available logic capacity was very limited and using run-time reconfiguration had been suggested to raise resource utilization. For example, the dynamic instruction set computer (DISC) [27] is capable to change its instruction set at run-time according to a running program. Consequently, only the currently used instructions are loaded on the FPGA which takes fewer resources than having a configuration providing all instructions at the same time. However, for decades implementing such systems required deep knowledge about the used FPGA architecture and has to follow an error prune difficult design flow. Furthermore, long configuration time was another crucial issue that detained the use of run-time reconfiguration (RC) in industrial applications.

With the progress in silicon process technology, logic capacity raised steadily while getting cheaper (and often more power efficient) at the same time. This

removed the pressure on the FPGA vendors to add better support for partial reconfiguration (PR) in their tools and devices. However, by heading towards huge 1M LUT devices (million look-up table FPGAs), things are changing dramatically at the moment. Note that we focus on SRAM-based FPGAs in this tutorial.

Here, the configuration time required to write tens of megabytes of initial configuration data is too long for many applications. For example, the PCIe host interface standard requires a device connected to the bus to answer within 100 ms after reset. This requires either a costly multi FPGA solution, changes in the software, or bootstrapping. In the latter option, only the time critical parts of the system (e.g., a PCIe interface core) will be configured at system start while leaving the configuration of the rest of the system to a second partial configuration step. Together with the advantages of resource saving (monetary cost and power consumption) FPGA vendors are now forced to enhance the support for run-time reconfiguration. In other words, partial reconfiguration will be available in the majority of future FPGA devices and corresponding tools.

This tutorial is devoted to: 1) PR design tools and 2) applications that substantially benefit from run-time reconfiguration. The first topic is presented in Section 2 that will introduce the special requirements on the physical implementation and that will give an overview on PR tools and implementation techniques. This is assisted by the case study of a reconfigurable CPU in Section 3. After this, it will be shown how run-time reconfiguration can improve resource utilization as well as system flexibility for different applications. Section 4 reveals a self-adaptive video processing platform that is capable of sharing reconfigurable regions by various modules arbitrary at run-time. Next, Section 5 presents a reconfigurable SQL database accelerator. After this, Section 6 and Section 7 give examples of how to reduce implementation cost while enhancing adaptability in SDR applications with the help of PR.

The here presented applications give only a short insight into the active research on implementing run-time reconfigurable systems and there are many more examples that demonstrate that the use cases of partial configuration are virtually unbound. For example, [14,3] demonstrate partial FPGA reconfiguration for image processing, [25,4] propose self-adaptive control systems, and [5,20] investigate to exploit partial reconfiguration for high performance computing (HPC).

## 2 Design Tools and Techniques for Partially Reconfigurable Systems

Partial run-time reconfiguration has to be supported by the FPGA and by the corresponding design tools. While for example all recent FPGAs from the vendor Xilinx support partial reconfiguration, this feature is only available for the Virtex series devices within the vendor tools. Moreover, this feature is not available by default and has to activated as a separate feature.

### 2.1 Terms and Definitions

A reconfigurable system is partitioned into two parts. 1) The part of the system that is always present (e.g., a memory controller or a soft CPU) is commonly called *static region* while 2) the part containing run-time reconfigurable modules is called *partial region*. If the static part of the system includes the logic to re-configure the device, this is called *self-reconfiguration*. All recent FPGAs from the vendor Xilinx provide an internal configuration access port (ICAP) for self-reconfiguration and no external pins are required to access the FPGA configuration. The same is announced for the new Stratix-V Series from Altera.

As shown in Figure 1, the partial region can be arranged in different configuration styles. In the easiest case, the partial region follows the *island style* approach that is capable of hosting one reconfigurable module exclusively per island. A system might provide multiple islands and if a set of modules can only run in one specific island this is called *single island style*. If modules can be *relocated* to different islands this is called *multi island style*

While the island style is ideal for systems where only a few modules are swapped, it might suffer *internal fragmentation*. This is a waste of logic resources that arises if modules with different resource requirements share the same island exclusively. This means, if a large module cannot be replaced by multiple smaller ones (to be hosted at the same time) the utilization of the reconfigurable region is getting weak. Internal fragmentation is improved by tiling reconfigurable regions using the one-dimensional *slot-style* or the two-dimensional *grid-style* approach. In this technique, a module occupies a number of tiles according to its resource requirements and multiple modules can be hosted simultaneously in a reconfigurable region. This is illustrated in Figure 1b) and Figure 1c)

Tiling the reconfigurable region is considerable more complex as the system has to provide communication to and from the reconfigurable modules and to determine the placement for a module. The latter one has to consider that FPGA resources are in general heterogeneous (i.e. there are different primitives like logic, memory, or arithmetic primitives on the fabric). Moreover, depending on the present module layout, a tiled reconfigurable region might not provide all free tiles as one continuous area. If this results in unused tiles, this overhead is called *external fragmentation*. External fragmentation can be removed by defragmenting the module layout which is called *compaction*. Note that some modules cannot be interrupted for compaction because a module would loose its internal state or it would not meet its throughput.

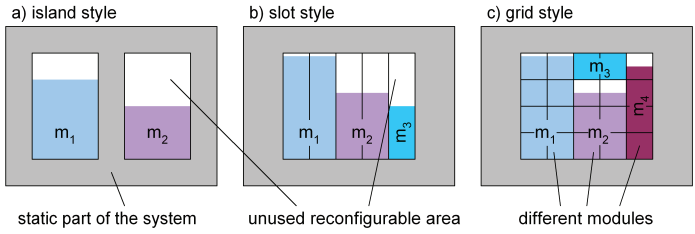



**Fig. 1.** Reconfiguration style.

### 2.2 FPGA Support for Partial Run-time Reconfiguration

Partial reconfiguration requires from the FPGA architecture that parts of the fabric can be overwritten without affecting other parts of system. Note that we imply that the static part of the system is active at any time (independent of a configuration process).

Different FPGA architectures show different behavior during the reconfiguration process. For example, older Xilinx FPGA families, including Virtex-II or Spartan-3 FPGAs, could only be reconfigured full column-wise, meaning that a full column of resources (logic, or routing) was affected when writing new configuration data to the device. This can cause side effects to the static system or other reconfigurable modules if they use resources in a column that can be reconfigured at run-time. On recent Xilinx FPGA architectures, the height of the columns had been reduced to the height of a clock region. This represents four block RAMs, four multiplier primitives, or 16-20 configurable logic blocks (CLBs), depending on the FPGA family. Note that the fabric of Xilinx FPGAs consists of a regular mesh of CLBs that each provide 8 look-up tables and an attached switch matrix to carry out the routing with surrounding CLBs. In some columns, the LUTs have been replaced with dedicated blocks such as memories while maintaining regularity of the mesh. There are several wires leaving each switch matrix that are named by their routing distance (e.g., single wires route one CLB further, double two CLBs, and so forth). By restricting reconfigurable tiles to span the height of full clock regions, side effects due to partial reconfiguration can be avoided on recent Xilinx FPGAs.

Altera announced to use mask mechanisms to control which part of the FPGA will be changed by reconfiguration. This technique should allow arbitrary reconfiguration without side effects to other active parts of a system [17].

### 2.3 Place & Route Constraints for Island Style Reconfiguration

For implementing a run-time reconfigurable system, we have to ensure that 1) a partial module uses only resources (logic & routing) that are not used by other parts of the system (another partial module or the static system) and 2) that identical wires of the FPGA fabric are used to connect partial modules over all physical implementation steps. This means that we have to constrain the routing to use a specific wire for crossing the module border for each signal bit of the module entity. The begin and end ports of these wires can be compared with plug-socket pairs on a printed circuit board. As additional constraints, we have to 3) activate all clock trees that are used by partial modules and 4) constrain the timing. This is mostly everything to consider and there is nothing mystic behind applying partial reconfiguration. The real difficulty stems from a lack in the design tools that, for example, provide no constraints to bind a signal to specific wires on the fabric. For instance, there is no constraint to bind a signal in a top level design that is responsible for the communication with a partial module to a specific wire that crosses the partial-to-static border.

To overcome this, macros called *bus macros* have been used in earlier partial design flows by the vendor Xilinx [15,28]. As shown in Figure 2.3 a), the signal to wire binding has been achieved by instantiating a macro consisting of a LUT

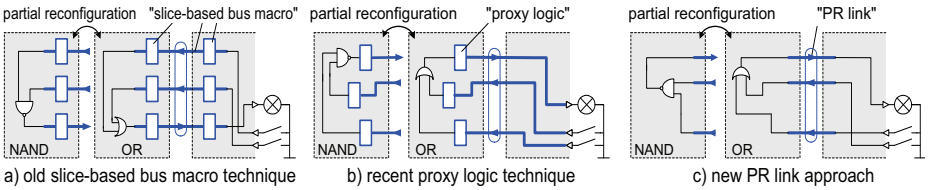



**Fig. 2.** Different approaches for constraining the signal to wire binding. In all three examples, a partial NAND module is exchanged by an OR module.

in the static part and a corresponding LUT in the reconfigurable region. By placing bus macros at a defined position on the partial module border, signals are bound to the internal macro wires.

The bus macro approach costs two LUTs per signal wire and additional latency. This was improved in the recent Xilinx vendor tools using an incremental partial design flow [29]. As shown in Figure 2.3 b), anchor LUTs in route-through mode (called *proxy logic*) are placed in the partial region for each signal crossing the partial module border. During the implementation of the static system, the partial interface signals are routed to the anchors. The partial modules are implemented incrementally from this static system without modifying any static routing. Consequently, the partial module interface wires are constrained by preserving the initial static routing.

The proxy logic approach costs one LUT per signal wire. Its main drawback is that the routing is different for each reconfigurable island. This prevents module relocation even if the islands provide an identical logic or memory layout. Moreover, changes in the static system will in general result in a different routing to the proxy logic. Consequently, all permutations of module type and placement position have to be rerouted on each modification in the static system. Hence, this approach is only suitable for systems of low complexity.

As a third approach, it is possible to prohibit selectively the use of wires by occupying them with the help of *blocker macros* [12]. Blocker macros instantiate within a defined region all logic primitives and use all available wires (or a selected set of wires) within this region. By selectively not blocking wires, *tunnels* can be drilled through a blocker such that only one possible routing path exists to route across the border to and from a reconfigurable module. By this, we force the router to bind a signal to the wires of the tunnel. By defining a tunnel in the partial region during the implementation of the static system and by defining a tunnel in the static region during the partial module implementation, each module interface signal is bound to a corresponding interface wire, called *PR link*. As shown in Figure 2.3 c), this allows module integration without logic overhead while permitting module relocation and an independent implementation of the static system or any partial module. The important difference between the proxy logic and the PR link approach is that the latter one constraints a signal to a specific wire rather than allowing the router to decide the signal to wire binding during the static system implementation.

For defining the placement of logic primitives and for specifying the timing requirements, the Xilinx vendor tools provide sophisticated constraints. Note that the definition of reconfigurable regions and module bounding boxes does not have to be rectangular. And because routing will not be interfered by recon-

figuration if the routing is overwritten with exact the same configuration data, it is possible to relax strict bounding box constraints for the routing. This can substantially improve performance and routability [13].

The clock signals are routed via dedicated clock networks. By connecting the clock to the blocker primitives located in the partial region during the static system implementation with the clock network, all clock network drivers are activated such that the reconfigurable modules can access one ore more clocks when loaded into the reconfigurable islands.

### 2.4 Communication Architectures for Slot and Grid Style Reconfiguration

When moving from an island style scenario to a slot or grid style reconfiguration scheme, extra precaution is needed to provide communication from and to the different partial modules loaded together into a reconfigurable region. In order to permit module relocation, we have to implement a *homogeneous communication architecture* that possesses an identical logic and routing footprint within each tile and at the tile borders. The base idea of such an architecture is shown in Figure 3 and related approaches have been presented as research work, e.g., [16,8,10]. The last work [10], is designed for high performance, flexible module placement and low implementation cost; and a system with 60 individual reconfigurable modules using grid style reconfiguration has been demonstrated in [11]. By arranging the logic and routing identical in each tile of a reconfigurable region and inside the modules, glitches on running bus transactions or data streams can be avoided during reconfiguration, regardless to the placement position. This is possible because the internal routing and logic of the communication architecture will never be changed at runtime.

### 2.5 PR Design Tools

The bus macro approach for Xilinx FPGAs has dominated the FPGA run-time reconfiguration community for more than a decade. However, Xilinx moved completely over to their new 4th generation PR flow based on proxy logic while

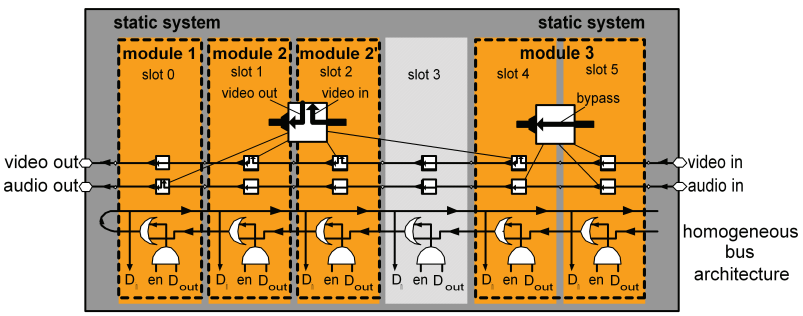



**Fig. 3.** Basic approach for a homogeneous communication architecture.

removing the support for bus macros. For this flow, Xilinx included some extra features in their flooplanning and constraint generation program PlanAhead. This includes resource budgeting, defining reconfigurable islands, and an automatic placement of proxy logic. While especially the last feature has substantially simplified the design flow, a lack of routing constraints prevents this flow to implement multi-island or slot and grid style reconfiguration as described in Section 2.3.

In addition to Xilinx, Altera (as another major FPGA vendor) has announced to support partial reconfiguration in their devices and design tools. According to [17], designing partial systems will be based on an incremental design flow very similar to the recent flow proposed by Xilinx. Consequently, these systems will include the same limitations (no module relocation, no static system to partial module decoupling during the implementation).

While a couple of research projects build tools on top of the Xilinx vendor PR tools, only little research was undertaken on developing independent alternatives. One alternative is the project ReCoBus-Builder [10]. This tool can generate constraints similar to PlanAhead, but it can also generate homogeneous communication architectures, as described in Section 2.4. In addition, the ReCoBus-Builder can constrain the routing by generating *blocker macros* as described in Section 2.3. The applications in Section 4, 5, and 6 have been implemented using the ReCoBus-Builder. As a further example, the tool OpenPR [24] reintroduces bus macros (see also Figure 2.3b)) into latest PlanAhead versions.

**Next Generation PR Tools** The preliminary purpose of PR design tools is to generate implementation constraints for the physical implementation of a system. Consequently, PR design tools have to generate constraints that are compatible with other tools following the constraints (e.g., place & route tools). Unfortunately, devices and design tools of the vendor Xilinx have changed that much that the concepts of the ReCoBus-Builder could not be easily shifted to recent devices and the support of that tool is limited to Spartan-3 and Virtex-II/II Pro FPGAs. Similarly, OpenPR is bound to Virtex-4 and Virtex-5 FPGAs.

For implementing run-time reconfigurable systems on recent FPGAs using latest vendor tools, a completely redesign of the ReCoBus-Builder is currently under development under the name GoAhead. This tutorial will announce its features and the tool will be available on the COSRECOS project website [2]. GoAhead provides a GUI (Fig. 4) and command script interface for floorplanning and macro placement that is similar to PlanAhead from Xilinx. The tool is shipped with a macro library containing various macros for different Xilinx FPGA families (Virtex-5/6/7 Spartan-6), including different *bus macros* and *connection macros*. A connection macro is basically a connection primitive (e.g., a look-up table) used to force the router to generate a routing path to or from this macro. GoAhead supports any reconfiguration style (Fig. 1) and the integration of modules using bus macros, PR-links (Fig. 2.3), or homogeneous communication architectures (Fig. 3). It can generate VHDL templates, UCF constraints (user constraints to be used with the Xilinx vendor tools), and routing constraints by generating blocker macros. The next section gives an example of how a reconfigurable system can be implemented using GoAhead.


**Fig. 4.** GoAhead GUI showing a Spartan-6 LX16 FPGA.

## 3 Reconfigurable Instruction Set Extensions

Changing the instruction set architecture (ISA) of a softcore CPU at run-time can substantially enhance performance and area at the same time. This has been demonstrated several times before (e.g., [27]) and even small instructions can gain high speed-ups. For example, if we consider a dedicated custom instruction for permuting all bits in a 32 bit operand, this would easily take a hundred cycles on a conventional CPU but would be only wiring, if implemented as a dedicated instruction. Implementing such instructions reconfigurable allows more instructions on less area. This involves some reconfiguration overhead that might be hidden by *configuration prefetching* in some systems. However, the real difficulty in implementing reconfigurable custom instructions is that a relatively large number of signals have to be connected to small modules. For example, a 32-bit instruction with two operands and one result requires a connection of roughly 100 signal bits (=wires); and assuming two LUTs per result bit, the instruction can be implemented in just eight CLBs on a Xilinx FPGA.

Figure 5 a) shows a simplified architecture of a CPU extended by four slots to host reconfigurable instructions. We will now reveal how a corresponding system can be implemented with the tool GoAhead. As sketched in Figure 5 b) for Xilinx Spartan-6 or Virtex-6 FPGAs, different wire resources have been used to connect two operands (with single and double lines) and the individual results (quad lines) for four adjacent slots. The wires have been chosen such that the operands and results can be connected at exactly the same relative position in each slot in order to permit module relocation. Modules may take more than one slot by using only one of the result vectors. Note that the input operands get swapped after each slot. This might require design alternatives, if operations are not commutative.

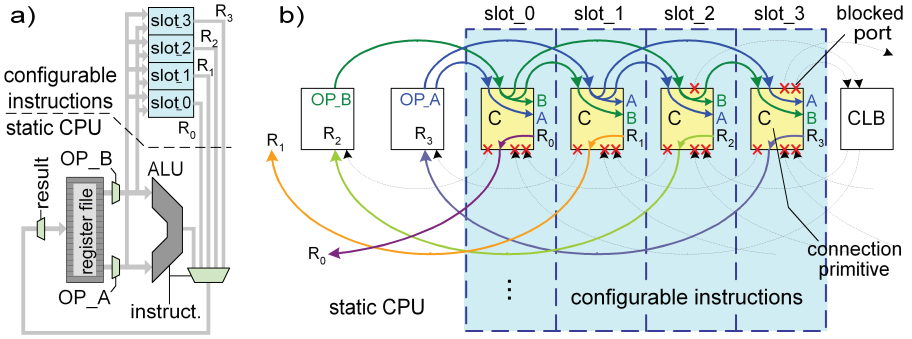



**Fig. 5.** Reconfigurable instruction set extension. a) RTL b) FPGA implementation. The implementation uses neatly single, double and quad lines that route one, two, and respectively four switch matrices (CLBs) further on Xilinx Virtex-6/Spartan-6 FPGAs.

On Xilinx Spartan-6 or Virtex-6 FPGAs we can connect four bits per two operands and one result per CLB. Figure 5 b) follows this mapping when assuming that each shown connection represents a bundle of four wires connected to a slice acting as the connection macro. Then a 32-bit instruction can be mapped in a slot as small as eight CLBs. The following GoAhead script places the connection macros, releases the ports used to route operands and results, generates a blocker, and writes instantiation code and user constraints for the Xilinx ISE tools. As can be seen, all constraints can be generated with only a few different commands. The X/Y coordinates are CLB tile coordinates, `WW4`, `EE2`, and `ER1` are names used by Xilinx for quad lines towards west, double lines towards east, and single lines towards east. The connection primitive `Connect4_S6_CI` is provided by GoAhead.

```
 1: AddSingleMacro MacroName=Connect4_S6_CI InstanceName=Slot0Inst0 Slice=SLICE_X19Y12;
 2: AddSingleMacro ...  # add 8 macros in each of the four slots
33: AddToSelectionXY X1=31 Y1=50 X2=34 Y2=57;  # select Slot0 and Slot1
34: DoNotBlockPort PortNameRegexp=WW4B; # release west quadline begin ports (result vectors)
35: DoNotBlockPort PortNameRegexp=EE2; # release east double line begin/end ports (operands)
36: DoNotBlockPort PortNameRegexp=ER1; # release east single line begin/end ports (operands)
37: AddToSelectionXY X1=35 Y1=50 X2=36 Y2=57;  # select Slot2
38: DoNotBlockPort ... # release used ports for slot2 and also slot3 according to Figure 5b)
50: PrintVHDLMacroInstantiation PortMapping=Sin:OPA,Din:OPB,Res:Res0,CLK:clk Filter=Slot0;
51: PrintVHDLMacroInstantiation ...  # generate VHDL instantiation code for all four slots
54: AddToSelectionXY X1=31 Y1=49 X2=38 Y2=57;  # select all four slots
55: BlockSelection OutFile=StaticBlocker.xdl  # block all remaining wire resources
56: PrintLocationConstraintsForPlacedMacros FileName=static.ucf;  # for ISE project
57: PrintPlacementProhibitConstraints FileName=static.ucf;  # prevent the Xilinx tools to
      # use any further primitive inside the current selection (i.e. the reconfig. area)
```

By placing the macros in a bottom-up order, we cause a connection of the operand and result vector signals also in a bottom-up order. This reduces congestion in modules using carry chain logic that propagates in the same bottom-up direction. By connecting four signals per CLB row, we incorporate that a carry chain processes also four bits per CLB row on Xilinx FPGAs. In order to generate the configuration bitstream for the static system, we run synthesis, technology mapping and placement in the Xilinx Vendor tools. After this, a batch script in-

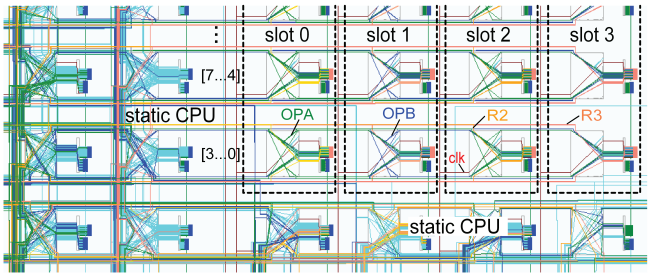



**Fig. 6.** FPGA Editor screen shot of a MIPS CPU providing four slots for reconfigurable custom instructions on a Xilinx Spartan-6 FPGA.

cludes the blocker into the design and runs the vendor router. Finally, we delete the blocker and generate the configuration bitstream. An FPGA Editor screen shot of the final system is shown in Figure 6.

The partial modules (i.e. the reconfigurable instructions) are implemented very similar to the static system, except that the connection macros are now placed left and right beside the used slots as a placeholder for the static system. We use blockers with released ports that result in an interface compatible to the static system. For the timing verification, GoAhead can compose netlists of the static system and any possible combination of placed modules. Finally, GoAhead generate partial bitstreams for a user defined region that can be directly written to any configuration port of the device.

As sketched in this section, GoAhead can be used for implementing very sophisticated reconfigurable systems. However, this requires still a considerable knowledge about the used FPGA architecture. The final version of the tool will provide wizards that hide most of the low level details of the fabric.

## 4 A Self-adaptive Reconfigurable Video Processing System

In this section, we present a system architecture for building partially reconfigurable *System-on-Chips* (SoCs), described in details in [21]. This architecture is exemplary applied for a smart camera system. FPGA-based embedded systems are of increasing importance especially in the signal and image processing domain. For instance, intelligent embedded systems for image processing, such as smart cameras, rely on FPGA-based architectures [23]. With the advantage of reconfigurability, we can envisage new designs with new and improved possibilities and properties, like an adaptive design which can adapt itself to a new operation environment. The static part of the system provides a CPU, the SoC infrastructure and the interfaces for the video input of the camera system. Most of the image processing algorithms, e.g., filtering, color transformation and detection, or visualization modules (called marker modules) are implemented as partially reconfigurable modules which can be dynamically loaded and unloaded at run-time.

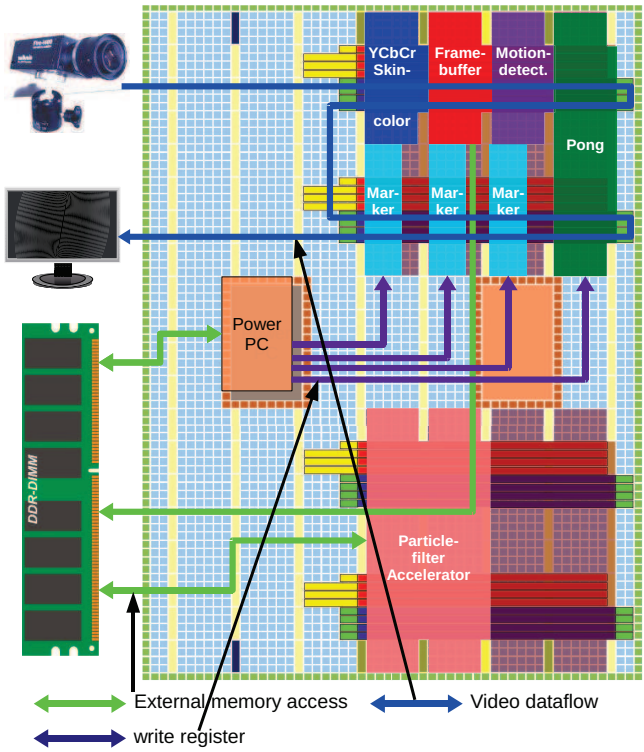



**Fig. 7.** System overview of the heterogeneous FPGA-based smart camera SoC platform consisting of CPU sub-system and reconfigurable area. Reconfigurable modules can vary in size and be freely placed, allowing a very good exploitation of the FPGA space.

### 4.1 Architecture

The system is implemented on the Xilinx Virtex-II Pro XUP board and consists of an embedded CPU sub-system including the external DDR-memory and the reconfigurable part (see Figure 7). In the following, these components and the communication interfaces between them are presented.

**Embedded CPU Sub-system** The main purpose of the software part on the embedded CPU is to control and manage the overall system. It contains high-performance peripherals, interfaces, and other IP cores. These are, e.g., a memory controller to provide access to an external RAM, a serial port interface for user commands, and a module for accessing the integrated reconfiguration interface of the FPGA. All components of the embedded CPU sub-system are connected by the main on-chip system bus, the *processor local bus* (PLB).

**Reconfigurable Area** The FPGA area is divided into a static and a dynamic part (see Fig. 7). The two marked areas on the right top and bottom compose the dynamic part of the system. Reconfiguration is only possible in the dynamic part

which contains a reconfigurable on-chip bus (*ReCoBus*) and *I/O bar* communication primitives to provide a communication infrastructure for dynamically loaded hardware modules. Both communication primitives part of the ReCoBus-Builder framework [10]. In the smart camera platform, the I/O bar is used to stream video data between the various reconfigurable processing modules. The modules can read and modify the video stream or generate additional output signals. To allow communication between the embedded CPU sub-system and the reconfigurable part, a PLB/RCB bridge translates the ReCoBus (RCB) protocol to the PLB protocol and vice versa. Using the ReCoBus and the bridge, the modules can be accessed from the CPU, e.g., to configure the module with memory-mapped registers. Furthermore, the modules have also direct access to the external memory (DMA). To allow high-speed data transfers between hardware masters and the memory controller, the bridge uses the *native port interface* (NPI) of the memory controller (provided by Xilinx).

### 4.2 Reconfigurable Modules

We implement several reconfigurable modules to tackle a wide spectrum of applications for our smart camera platform. In this section, we present some of these modules.

The *skin color detection* is implemented as a hardware slave module that reads the color values from the I/O bar and marks them as *skin* or *non-skin* by comparing them with a color template. We have implemented modules for RGB and YCbCr color spaces. The classification is written as an additional signal (skin color bit) onto the I/O bar together with the unmodified video stream.

The *filter module* is a sliding-window image processing filter. The current implementation supports a 3x3 filter matrix. To access different image lines, the module stores two lines in a BRAM-FIFO. The coefficients are stored in CPU accessible registers. Therefore, a module can be configured for different filter functions, for example, with the coefficients of a Sobel filter which can be used for edge detection.

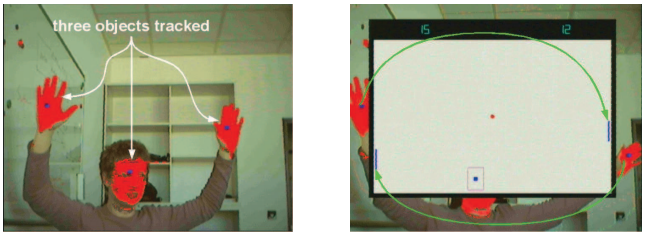
The *framebuffer* hardware master module is implemented to store the current input image. This is done by double buffering the images in the on-chip memory via the ReCoBus using the NPI interface. We use 32 Bit for storing one pixel, with 24 Bit for the input RGB values and the remaining 8 Bit free for classification results, e.g., the skin color bit.

The *particle filtering framework* is partitioned into a software and hardware part. The software part performs the sampling and applies the motion model. The hardware part is used as a co-processor to perform the evaluation steps.

The *motion detection module* compares the pixel values of two subsequent images to detection motion. Like the skin color detection module, the result (motion/no motion) is written as an additional signal onto the I/O bar.

The *pixel marker module* colors classified pixel or regions with a specified color. The classification of the pixel is signaled to the marker module with additional I/O bar signals. The color can be configured by a register interface.

An embedded design for tracking human motion is implemented as an example application to show the flexibility of the proposed platform. The idea is




(a) The particle filter tracking three objects.

(b) The object tracker used to play a pong game.

**Fig. 8.** The smart camera tracks three image regions (a person's head and hands). The tracked hand positions are directly used to control the paddles of the video game.

to detect and track skin-colored image regions, which is done by applying particle filtering. The current implementation makes it possible to track up to three image regions. One marker module is used per region tracker. A simple *tennis game* is implemented on top of this application, which can be directly controlled by the hands of a person, using the results of the tracker (see Fig. 8).

## 5 Partial Reconfiguration for SQL Query Processing

This section describes how dynamic partial reconfiguration can be used for SQL query processing for large databases. There exist already static FPGA approaches, e.g., *Glacier*, a query-to-hardware compiler [18], and Netezza's *FAST*-engines [7]. Both approaches have a lack of flexibility: *Glacier* has to run a full synthesis for every new incoming query and Netezza's *FAST*-engines have a fixed pipeline with no possibility to reorder operations.

We cover restrictions (WHERE-clauses) and allow different data types for table attributes as well as different operations on these data types. Restrictions are Boolean expressions which are used to filter out tuples from a table. Tuples remain in the result table if they are evaluated to *true* regarding the restriction, otherwise they are omitted. Figure 9 shows an overview of the proposed SQL accelerator technique.

We offer a module library with different operators, which can be used to assemble a pipeline at run-time by using partial reconfiguration, to form different restrictions. We support integer types up to 32 bit and fixed-length strings. Furthermore, we offer arithmetic-logical operators ($+$, $-$, $*$, $AND$, $OR$, $NOT$, $XOR$, $NAND$, $NOR$) and comparisons ($<$, $\leq$, $=$, $\neq$, $\geq$, $>$). The latter ones can be used on integer attributes as well as string attributes. The pipeline processes tuples one after another and computes the restriction result for each tuple. The supported operators only rely on tuple data itself, thus we can evaluate the restriction for each tuple itself independent from other tuples.

We use I/O bars to pass data from module to module. Furthermore, we use them to configure modules which are plugged onto the bars. Usually, tuples are wider than the data bus width of the I/O bars, which is 32 bit in our system, therefore tuples are divided into several chunks and tuples are processed

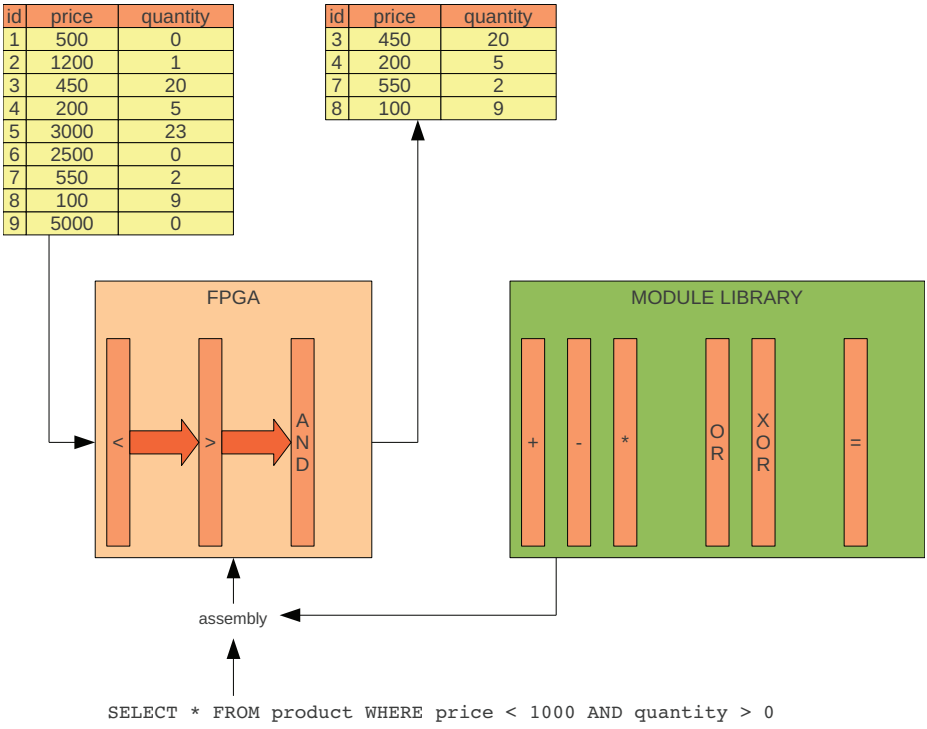



**Fig. 9.** FPGA configured with partial modules to perform a restriction (WHERE-clause). The partial modules are assembled to a pipeline for high-throughput.

chunk-by-chunk in a fully pipelined manner. Modules are configured with the information about tuple sizes and the chunk indices of their operands. Furthermore, they are told where to put their results in the tuple stream. The modules are capable of doing attribute-attribute operations as well as attribute-constant operations, i.e., either both operands are part of the tuple or one operand is a constant value which is configured during configuration of the module. Figure 10 shows an example of an arithmetic-logical module which performs an addition.

We append spare chunks to the tuples, thus the modules can place their results inside the tuple stream. By looking up the result of the last module in the pipeline, which is either *true* or *false*, we can decide whether we keep a tuple in the result table or not.

With the use of dynamic partial reconfiguration and a presynthesized module library, we are able to switch the functionality during run-time, thus we can execute queries with different restrictions one after another without any further synthesis, which was the drawback of *Glacier*. Furthermore, it would be possible to implement further modules to support more operations, e.g., projections, or more data types like floating point. Thus, we could switch the operation order of such SQL operations, which is not possible with Netezza's *FAST*-engines because their pipeline is fixed. We used the XUP Virtex-II Pro Evaluation Board for

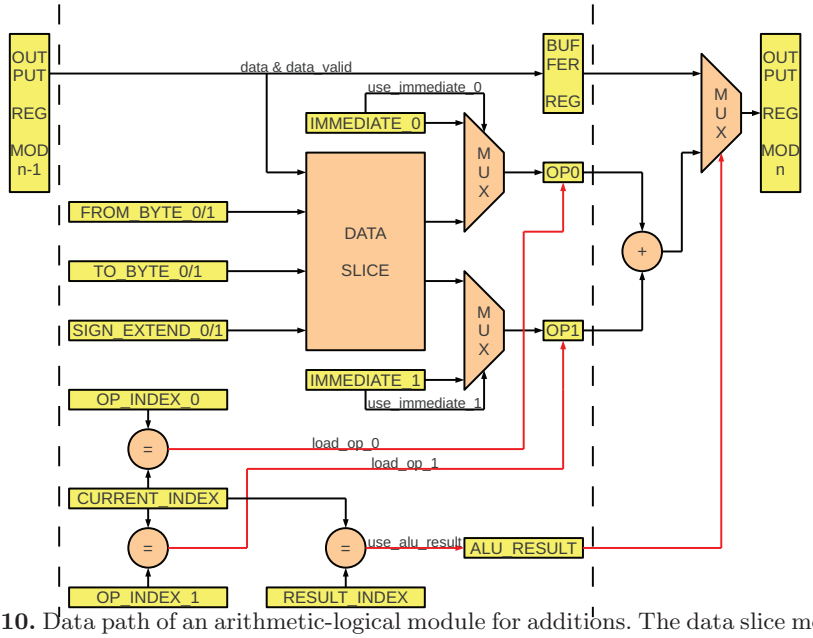



**Fig. 10.** Data path of an arithmetic-logical module for additions. The data slice module is responsible for truncating or padding attributes to the size of a 32-bit chunk.

prototyping. Our system runs with 100 MHz, thus we reach a total throughput of $400\frac{MB}{s}$ due to our pipelined design.

## 6 Parameter Controlled Software Defined Radio Platform

Software Defined Radio (SDR) is promoted by the the Wireless Innovation Forum which describes it as *"[. . . ] a collection of hardware and software technologies where some or all of the radio's operating functions (also referred to as physical layer processing) are implemented through modifiable software or firmware operating on programmable processing technologies."* [26]. The idea behind SDR is to extend the range of flexibility in radio development and to provide the ability to adapt to future wireless standards, to add new features and capabilities to existing infrastructure or to fix corrupt designs which causes system misbehaviors. Such a system has been developed to enhance the flexibility of the hardware design of a wireless tracking technology by utilizing the benefits of the ReCoBus technology and partial reconfiguration of FPGAs. The system offers generic digital signal processing modules for mixing, filtering and correlation tasks. The modules can be placed as pleased and the necessary parameters like filter coefficients are configured by a SoC infrastructure.

### 6.1 Architecture

The system is implemented on custom hardware platform which features a Xilinx XC2VP70 FPGA. Furthermore, the hardware provides an optical datalink to a

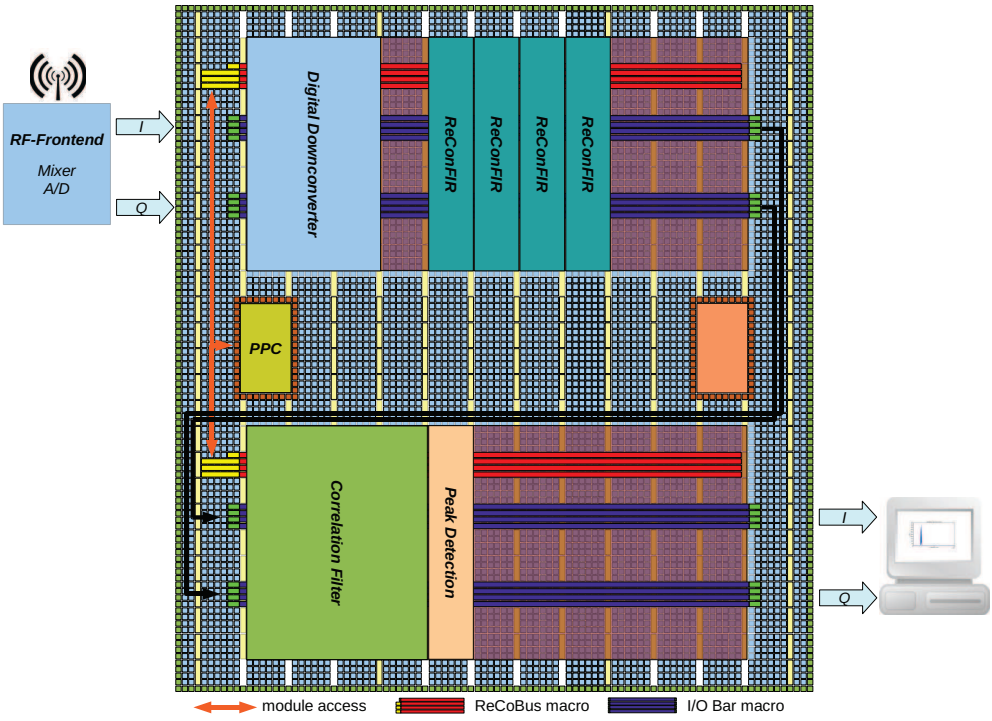



**Fig. 11.** SDR platform consisting of a controll CPU sub-system and two reconfigureable areas. The example shows the configuration used to track individual radio transmitters.

RF frontend and a 64 Bit PCI-X interface which is used to retrieve the processing results for further usage on the host system (see Figure 11).

One of the embedded PowerPCs of the Virtex-II Pro FPGA is used to create a configuration master which duty it is to communicate with configured modules via the ReCoBus macro to set module specific parameters.

The design features two large reconfigureable areas which can be used to freely place the designed modules. The areas are crossed by two I/O Bars with each providing a 32 Bit interface to offer a reasonable amount of signal bits. Above the I/O Bars, an on-chip bus structure has been place to configure and steer the modules (the ReCoBus macro). Furthermore, the reconfigureable areas feature two clock signals driven by low skew clock nets which can be utilized by modules independently of their placement.

### 6.2 Reconfigureable Modules

We implemented several reconfigurable modules for SDR applications. One property that applies to all modules is that the signal bit width is limited due to the finite size of the I/O Bars. As a consequence, all modules include an adjustable bit slicer to meet the specifications.

The *digital down converter module* divides the incoming baseband signal into subchannels and down samples the signal to the absolute necessary rate. As a new approach and as an extension to the existing ReCoBus design flow, this module is designed by structural description in *Xilinx System Generator* (XSG), which is a blockset add-on for the Matlab Simulink environment. This procedure enables the user to rapidly design complex modules.

The *mixer module* comprises a *Direct Digital Synthesizer* (DDS) to generate the sine and cosine waveforms and a complex multiplier to shift the signal from one frequency to another. The *ReConFIR* named module is designed to offer the user a generic 8-tap FIR filter. Necessary parameters, including the filter coefficients and the expected sampling rate of the FIR filter, are configured by the SoC. As a special feature, this module can be cascaded with other ReConFIR module instances to create an ReConfigureable n-tap FIR filter of variable length *n*. The *correlation filter module* is designed to meet the use case specific requirements regarding the type and amount of coefficients. Nevertheless, these can be configured via the PowerPC platform to change the tracked signals. The *peak detection module* searches for peaks in the correlation results and indicates its results by setting a signal.

### 6.3 Summary

The system benefits significantly from the discussed technologies by adding the ability of reusing design elements, changing key characteristics such as the used frequency band or the tracked signal sequences without the need to design and configure a completely new design. A big variety of use cases are imaginable besides the one presented. The static platform can not only be used for receiver but also for transmitter designs by making use of the already build generic modules and additional system specific modules. Especially pulse-based transmission systems (like the mentioned tracking technology) offer a great margin for run-time reconfiguration because of the idle time between two bursts.

## 7 An SNR-Adaptive Cognitive Software-Defined Radio using Partial Reconfiguration

In mobile reception scenarios, the signal-to-noise ratio (SNR) of a receive signal can strongly vary over time. In situations where the signal is weak it might be necessary to spend more computational effort in decoding the signal to get a better performance. On the other hand, when the receive signal is strong with respect to the noise, it might be beneficial to reduce the decoding complexity as the reception quality is sufficient for the target application. Bearing this in mind, the use of PR of FPGA resources enables to change the complexity of one component inside the SDR chain according to the perceived SNR while keeping the other components active. Thus, a PR-based design may give a benefit over a static design, due to the hardware reusability in the reconfigurable area.

In the following sections we will introduce a PR use case for broadcast receivers, where the algorithms of an FM radio receiver chain will be adopted according to the estimated signal-to-noise ratio. The reconfigurable FM receiver

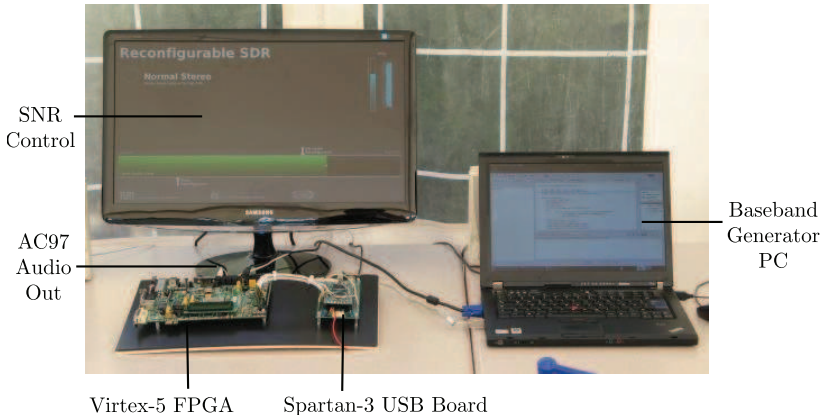



**Fig. 12.** The reconfigurable broadcast FM receiver demo system.

prototype was developed at the Technical University Munich as a simple case study to demonstrate the concept for SNR-adaptive cognitive radios and to serve as a template for further investigations of more complex applications.

### 7.1 Demonstration Platform

Our SNR-adaptive demo system implements a reconfigurable FM-RDS stereo broadcast receiver together with an SNR estimation stage, where the FM multiplex decoder can be reconfigured according to the estimated SNR [19]. The SNR is defined as carrier-to-noise ratio, reflecting the FM carrier signal power divided by the noise power.

The demonstration platform is shown in Figure 12 and consists of a PC, a Xilinx Spartan-3 FPGA and a reconfigurable Xilinx Virtex-5 FPGA (XC5VSX50T). The PC generates two complex baseband signals, each at a sample frequency of 500 kHz and transmits the data to the Spartan-3 FPGA via USB. The reconfigurable Virtex-5 device reads the data from a 16 bit parallel GPIO interface and processes it internally. The PC generates modulated FM stereo broadcast signals including Radio Data System (RDS) services and the SNR of these signals can be varied by adding white Gaussian noise to the respective stream.

On the reconfigurable Virtex-5 FPGA two FM baseband signals are received, decoded and the SNR is estimated. According to the estimated SNR at the receiver, the FM multiplex (MPX) decoding routines are adopted. For example, in case the SNR is very low, the receiver can either increase the computational effort and return a stereo audio signal which is more acceptable in quality or decrease the computational effort by switching to monaural decoding. In case the SNR is very high and the signal is very strong, the receiver can use low complexity demodulation algorithms while still getting a sufficient audio quality.

In the following sections, the receiver flow graph and the partitioning of the receiver chain on the FPGA are discussed.

### 7.2 FM Receiver Signal Flow and MPX Decoding

The flow graph of the FM receiver chain is shown in Figure 13. The receiver chain can be subpartitioned into three main parts, i.e. the mono decoding audio

part, the additional logic for stereo decoding and the RDS decoder. The mono signal combines the sum of the left and right audio signals. The stereo signal consists of the difference of the left and right audio signals. It is located at a frequency of 38 kHz using amplitude modulation with suppressed carrier. In order to coherently demodulate the stereo signal, the carrier has to be reconstructed. This is done by using the 19 kHz pilot tone which is extracted by a PLL in the stereo decoder part.

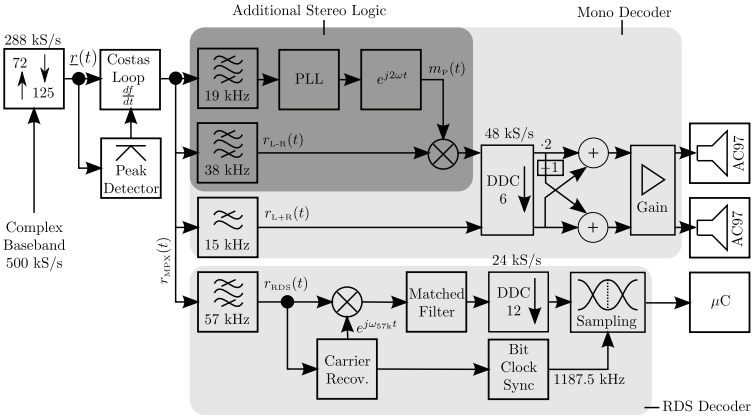



**Fig. 13.** FM receiver signal flow and partitioning. The grey background highlights the three different PR modules for MPX decoding.

In the next section, the partitioning of the decoder modules of the FM receiver chain for the implementation on the FPGA is presented. The module-based PR flow was used for the design of the reconfigurable partitions (c.f. [29]). The bitstreams for the different configurations are stored on a fast external on-board DDR2-Memory (max. 6.4 GB/s) and loaded on demand by the configuration control block.

### 7.3 Virtex-5 FPGA Partitioning

The FPGA configuration comprises one static and two reconfigurable partitions (reconfigurable islands). The static partition includes a Microblaze microcontroller, a multiplexed Costas loop for FM demodulation and two SNR estimation stages. The reconfigurable partitions are used for the demodulation of the respective MPX signal. Each partition can hold one of the following demodulator types: *Stereo* audio demodulator, *Mono* audio demodulator and an *RDS* demodulator. The number of logic elements of the partition was chosen with respect to the most complex design, i.e. the stereo demodulator. All configuration permutations are possible, e.g. the receiver can have two stereo demodulators, or one mono and one RDS demodulator, two RDS demodulators etc. The partitioning and signal flow of the demo platform is depicted in Figure 14.

In the figure, the reconfigurable partitions are denoted as *Partition A* and *Partition B*. Each partition can be reconfigured individually without interrupting

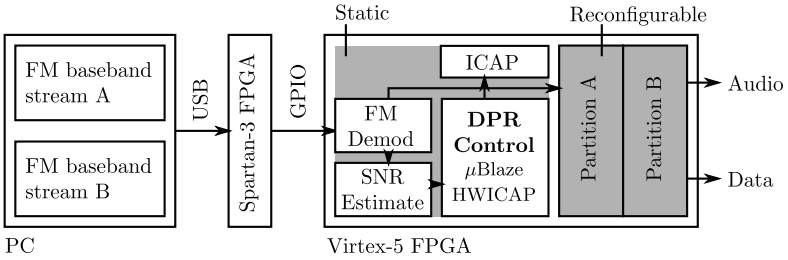



**Fig. 14.** Signal flow graph of FM demonstrator system. The Virtex-5 FPGA is used for FM demodulation and RDS decoding and comprises two reconfigurable partitions.

the other. Modules cannot be relocated and have been implemented separately for each partition (single island reconfiguration style). The microcontroller evaluates the estimated SNR values and is able to trigger a reconfiguration of partition A or B if the SNR reaches a certain threshold. The reconfiguration is done by reading the partial bitstreams from the external memory and writing them to the HWICAP module over the PLB. While the FM-MPX decoding chain is reconfigured via PR, the SNR estimation and the FM signal demodulation are constantly active in the static part of the device.

Either one of the three presented MPX decoder modules or an empty bitstream can be written to the reconfigurable FPGA partitions A and B. All configurations strongly differ in the number of slices, BRAMs and DSP multipliers as depicted in Table 1.

| MPX Decoder | Slices | DSP48 | BRAMs |
|---|---|---|---|
| Stereo audio decoder | 804 | 9 | 3 |
| Mono audio decoder | 458 | 2 | 2 |
| RDS data stream decoder | 503 | 6 | 5 |
| Empty (no decoding) | 0 | 0 | 0 |

**Table 1.** MPX decoder resource overview.

The stereo decoder is the most complex decoding branch, followed by the RDS decoder and the monaural audio decoder. In case one of the received signals is too noisy to demodulate, the respective partition can be replaced. If the noise power increases above a level where decoding is not feasible anymore, the MPX decoder in question is replaced by an empty bitstream.

The trigger for the reconfiguration is given by the Microblaze CPU. The reconfiguration conditions are presented in the following paragraphs.

### 7.4 Reconfiguration Conditions

For mono broadcasts our experiments have revealed that the audio distortion at SNRs below 4 dB is so strong that it is unbearable for the listener. In this case, the mono decoder will be removed from the active partition after the received signal has fallen below this threshold.

For stereo broadcasts the SNR must be approximately 21 dB above the mono threshold [22]. This is due to the fact that in FM the power spectral density of the demodulated MPX signal increases quadratically as the frequency increases [9]. Since the stereo difference signal is located at an intermediate frequency of 38 kHz it is more prone to noise than the monaural sum signal at DC. Thus, in case of stereo broadcasts it is feasible for the receiver to switch from stereo to mono if the SNR drops below 25 dB. Similarly, with our decoder implementation the SNR threshold for decoding RDS with a bit error rate below $10^{-3}$ is reached at an SNR of approximately 25 dB. Below that threshold, the RDS decoder is replaced with an empty bitstream in order to reduce the dynamic power consumption.

Hence, if the SNR estimator signals that the SNR has fallen below a certain threshold, the Microblaze CPU initiates a PR of the FM multiplex decoder to become more or less complex. The SNR-thresholds for the different reconfigurable partitions are summarized in Table 2.

| MPX Decoder | SNR Region |
|---|---|
| Stereo audio decoder | $\gamma \geq 25$ dB |
| Mono audio decoder | 4 dB $\leq \gamma < 25$ dB |
| RDS data stream decoder | $\gamma \geq 25$ dB |
| Empty audio (no decoding) | $\gamma < 4$ dB |
| Empty RDS (no decoding) | $\gamma < 25$ dB |

**Table 2.** SNR regions for PR partitions. $\gamma$ denotes the carrier-to-noise ratio in dB.

With the presented configuration conditions, the resources inside the reconfigurable region can be traded with respect to the actual requirements. An important fact is that by using PR and by regulating the amount of dynamic logic on the device, the dynamic power consumption of the receiver can also be regulated according to the user constraints.

### 7.5 Summary

The SNR in mobile reception scenarios is a function of time and vicinity. With PR of FPGAs the logic occupation of a mobile SDR receiver can be adopted to the actual requirements. The MPX decoders of the twin-tuner FM receiver presented in the analysis can be modified independently during the runtime. Thus, PR enables more degrees of freedom for cognitive SDRs on FPGAs.

Nowadays the reconfigurable region must be chosen with respect to the largest configuration, which might cause fragmentation of reconfigurable areas. In the future, the fragmentation could be reduced by sub-partitioning the reconfigurable partitions as proposed in [6]. However, although supported by the FPGA fabric, at the moment PR sub-partitions and module relocation are not supported by the Xilinx software suite, for this reason, we will investigate alternatives, such as GoAhead.

## 8 Conclusion

In this paper, we demonstrated promising use cases for partial reconfiguration on FPGAs as well as corresponding design tools. This started from tiny reconfigurable modifications of a CPU, over complex video processing and database acceleration up to two software defined radio applications. Common for all these applications is that they benefit from being able to relocate modules to different positions and to pack multiple modules together into a shared reconfigurable region. While this is not well supported by the tools from the major FPGA vendors, we showed upcoming alternatives developed in academia.

## Acknowledgment

This tutorial is joint work that is supported by different institutions. This includes the Norwegian Research Council founding the project Context Switching Reconfigurable Hardware for Communication Systems (COSRECOS) ([2]), under grant 191156V30 and the Bundesministerium für Wirtschaft und Technologie for supporting the project Programmable Telematic On-Board Radio (PROTON) ([1]) under Grant 10 P 8012B

## References

1. Programmable Telematic On-Board Radio (PROTON), `http://www.proton-plata.fr`
2. Project website: *Context Switching Reconfigurable Hardware for Communication Systems*, `http://www.mn.uio.no/ifi/english/research/projects/cosrecos/`
3. Claus, C., Stechele, W., Kovatsch, M., Angermeier, J., Teich, J.: A Comparison of Embedded Reconfigurable Video-processing Architectures. In: International Conference on Field Programmable Logic and Applications (FPL). pp. 587–590 (2008)
4. Commuri, S., Tadigotla, V., Sliger, L.: Task-based Hardware Reconfiguration in Mobile Robots Using FPGAs. J. Intell. Robotics Syst. 49, 111–134 (June 2007)
5. El-Araby, E., Gonzalez, I., El-Ghazawi, T.: Exploiting Partial Runtime Reconfiguration for High-Performance Reconfigurable Computing. ACM Trans. Reconfigurable Technol. Syst. 1, 21:1–21:23 (January 2009)
6. Feilen, M.: Concept and Design of an SNR-adaptive DRM+/FM Receiver using Dynamic Partial Reconguration (DPR) of FPGAs. 11th Workshop Digital Broadcasting (September 2010)
7. Francisco, P.: The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics. Tech. rep., IBM (2011), `http://www.netezza.com/documents/whitepapers/Netezza_Appliance_Architecture_WP.pdf`
8. Kalte, H., Porrmann, M., Rückert, U.: System-on-Programmable-Chip Approach Enabling Online Fine-Grained 1D-Placement. In: Proceedings of the 11th Reconfigurable Architectures Workshop (RAW). pp. 141–146. New Mexico, USA (2004)
9. Kammeyer, K.D.: Nachrichtenübertragung. B.G. Teubner, Reihe Informationstechnik, Stuttgart, Deutschland, 4 edn. (Mar 2008)
10. Koch, D., Beckhoff, C., Teich, J.: ReCoBus-Builder– a Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for FPGAs. In: Proc. of Int. Conf. on Field-Progr. Logic and Applications (FPL). pp. 119–124 (Sep 2008)

11. Koch, D., Beckhoff, C., Teich, J.: Minimizing Internal Fragmentation by Fine-grained Two-dimensional Module Placement for Runtime Reconfigurable Systems. In: 17th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE Computer Society, Napa, CA, USA (Apr 2009)
12. Koch, D., Beckhoff, C., Torresen, J.: Zero Logic Overhead Integration of Partially Reconfigurable Modules. In: Proceedings of the 23rd symposium on Integrated circuits and system design - SBCCI '10. p. 103. ACM Press (2010)
13. Koch, D., Torresen, J.: Routing Optimizations for Component-based System Design and Partial Run-time Reconfiguration on FPGAs. In: Proc. of Int. Conference on Field-Programmable Technology (ICFPT). pp. 460–464. IEEE (2010)
14. Krill, B., Ahmad, A., Amira, A., Rabah, H.: An Efficient FPGA-based Dynamic Partial Reconfiguration Design Flow and Environment for Image and Signal Processing IP Cores. Image Commun. 25, 377–387 (June 2010)
15. Lysaght, P., Blodget, B., Mason, J., Young, J., Bridgford, B.: Invited Paper: Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs. In: 2006 International Conference on Field Programmable Logic and Applications (FPL). pp. 1–6. IEEE (2006)
16. Mak, T.S.T., Sedcole, N.P., Cheung, P.Y.K., Luk, W.: On-FPGA Communication Architectures and Design Factors. In: Proceedings of the 16th International Conference on Field Programmable Logic and Applications (FPL). pp. 1–8 (2006)
17. Mark Bourgeault: Alteras Partial Reconfiguration Flow (2011), available online: http://www.eecg.utoronto.ca/~jayar/FPGAseminar/FPGA_Bourgeault_June23_2011.pdf
18. Müller, R.: Data Stream Processing on Embedded Devices. Ph.D. thesis, ETH Züich (2010)
19. Münch, D.: Receive signal dependent adaption of an FPGA-based Software Defined Radio receiver system. Master's thesis, TUM (2011)
20. na, M.S., Patel, A., Madill, C., Nunes, D., Wang, D., Chow, P., Wittig, R.: MPI as a Programming Model for High-Performance Reconfigurable Computers. ACM Trans. Reconfigurable Technol. Syst. 3, 22:1–22:29 (November 2010)
21. Oetken, A., Wildermann, S., Teich, J., Koch, D.: A Bus-based SoC Architecture for Flexible Module Placement on Reconfigurable FPGAs. In: Proceedings of the International Conference on Field Programmable Logic and Applications (FPL). pp. 234–239. Milan, Italy (Aug 2010)
22. Schad, F., Steil, A.: Laboruntersuchung über Versorgungskriterien für eine UKW-FM Monoabstrahlung (2008)
23. Schlessman, J., Chen, C.Y., Wolf, W., Ozer, B., Fujino, K., Itoh, K.: Hardware/software co-design of an FPGA-based embedded tracking system. In: Proceedings of CVPRW. p. 123 (2006)
24. Sohanghpurwala, A.A., Athanas, P., Frangieh, T., Wood, A.: OpenPR: An Open-Source Partial-Reconfiguration Toolkit for Xilinx FPGAs. In: Proc. of the IEEE Int. Symp. on Parallel and Distr. Processing Works. (IPDPSW). pp. 228–235 (2011)
25. Toscher, S., Reinemann, T., Kasper, R.: An Adaptive FPGA-Based Mechatronic Control System Supporting Partial Reconfiguration of Controller Functionalities. In: Pro. of the 1st Conf. on Adaptive Hardw. and Syst. (AHS). pp. 225–228 (2006)
26. Wireless Innovation Forum: What is Software Defined Radio, available online: http://www.wirelessinnovation.org/assets/documents/SoftwareDefinedRadio.pdf
27. Wirthlin, M., Hutchings, B.: A Dynamic Instruction Set Computer. In: Proceedings IEEE Symposium on FPGAs for Custom Computing Machines. pp. 99–107
28. Xilinx Inc.: Two Flows for Partial Reconfiguration: Module Based or Difference Based (May 2002), available online: www.xilinx.com/support/documentation/application_notes/xapp290.pdf
29. Xilinx Inc.: Partial Reconfiguration User Guide (2011), rel 13.2