Chapter 5
Reconfigurable CPU Instruction Set Extensions

Swapping just small fractions of the configuration of an FPGA can be very beneficial in many applications. This is in particular useful for reconfiguring the instruction set of embedded soft core processors. This is highly relevant for software driven design flows. Here, the system is initially implemented as far as possible in software (which is faster to accomplish than hardware development). By profiling the application, hot spots will be identified and kernals will be implemented for the FPGA for acceleration until performance requirements are met. There are several methodologies to integrate such accelerator modules. This ranges from small CPU instruction set extensions to large and fully autonomous modules that work concurrently with the CPU.

In this chapter, we will investigate how CPU instruction set extensions can be used efficiently with the help of partial runtime reconfiguration. The base idea of extending a CPU with exchangeable instructions is sketched in Fig. 5.1. Custom instructions access the register file in the same way as the ALU. By decoding unused instruction in the CPU ISA (instruction set architecture) a multiplexer may select between normal ALU operation or one or more user defined instructions.

Softcore CPUs with statically implemented custom instructions are well supported. For example the NIOS-II CPU from Altera can be easily extended with custom instructions when using the SOPC builder wizard of the Quartus design tools. Similarly, Xilinx allows to add custom hardware to thier Microblaze softcore CPU using FSL ports. These ports provide basically a streaming port interface between the Microblaze core and the custom hardware. However, for implementing *runtime reconfigurable* custom instructions, the support is weak, hence omitting this powerful opportunity.

In the following section, we will firstly demonstrate that commonly used techniques, like the Xilinx bus macro approach or the recent proxy logic technique is not well suited for integrating custom instructions. After this, in Sect. 5.2, we will demonstrate for a reconfigurable soft core processor that instructions can be integrated into the system without causing any additional logic overhead for the communication. In Sect. 5.3, we reveal how such systems can be easily implemented with the tool *ReCoBus-Builder*. Rather than providing reconfigurable islands, we

D. Koch, *Partial Reconfiguration on FPGAs: Architectures, Tools and Applications*,
Lecture Notes in Electrical Engineering 153, DOI 10.1007/978-1-4614-1225-0_5,

© Springer Science+Business Media New York 2013

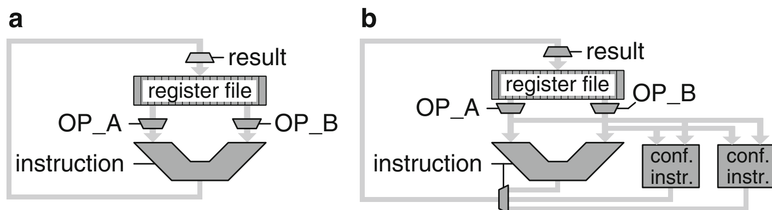

Fig. 5.1 (**a**) Example of a CPU data path. (**b**) Extension of the ALU with multiple configurable instructions. The modules are connected to the operands from the register file and a multiplexer selects between the different results

will integrate multiple custom instructions in a slot-based fashion. Finally, in Sect. 5.4, an experimental evaluation of a system providing a MIPS CPU extended to support reconfigurable custom instructions will be presented.

5.1 On-FPGA Communication for Custom Instructions

One basic problem to be solved in the design of partially reconfigurable systems is to constrain the routing of the interface signals for a partial module during its physical implementation. As introduced in Chap. 2, there are several ways to accomplish this. However, when considering a custom instruction set extension, as shown in Fig. 5.1, we have to consider that a relatively large wire count is required for connecting a relatively small reconfigurable area. For example, if we consider a bit permutation function, we have to connect 32 wires towards an island hosting the permutation accelerator and an additional 32 wires for the result. Note that this example would not demand any logic on the FPGA as a permutation is basically wiring on the FPGA. But still, as compared to a software implementation of a permute function, we can easily save a hundred or more assembly instructions, even if fully unroling the function. Or, if we consider a 64-bit XOR gate (over both operands) to be hosted in the same reconfigurable island, it requires two times 32 wires for connecting the input operands. But also in this case, we need only 21 4-input LUTs (e.g., on a Xilinx Virtex-II FPGA) or 13 or 6-input LUTs on a (Xilinx Spartan-6 FPGA) for implementing the 64 bit XOR gate. Again, this instruction would save about a hundred instructions per call of the function. In other words, for some programs, we could gain a substantial speed-up by just adding little additional logic. And by making this configurable, we could host virtually an infinite amount of different accelerators for supporting various software tasks.

When implementing such custom instructions with slice-based bus macros, as illustratrated in Fig. 5.2a, it takes two LUTs per signal wire only for providing the accelerator connection. If we consider in total 100 wires for linking two times a 32-bit operand, a 32-bit result vector and a few additional signals, The overhead is 200 LUTs. This is roughly 10 × more than actually needed for the XOR gate! Moreover,

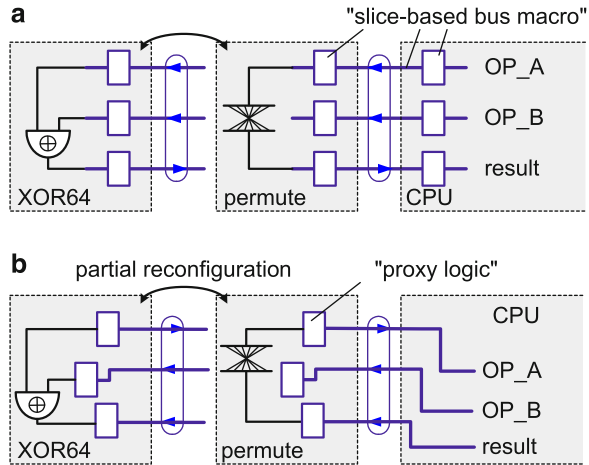

Fig. 5.2 Integrating custom instructions using (**a**) Xilinx bus macros, (**b**) proxy logic

the look-up tables constitute not only a logic overhead, but also a latency overhead which is roughly 0.4 ns on a Virtex-II FPGA per LUT. Finally, adding LUTs for the communication can negatively impact the placement of both, the static system and the partial modules. For example, a placed bus macro LUT interrupts carry chains and it can further force to spread a module over more area.

With the recent proxy-logic approach, the situation has improved, as shown in Fig. 5.2b. However, it still needs 100 LUTs for the communication. Again this is pure overhead in terms of resources and latency. And as explained in Sect. 2.4.5 on Page 71, the proxy logic approach is not well suited to implement systems with many different reconfigurable modules.

At this point, someone might think to use static only implementations instead, if custom instructions are that small. This is probably the better option for very few instructions. With a rising number of instructions, the CPU gets larger and consequently slower. When assuming the simplified diagram of a CPU datapath in Fig. 5.1a, the ALU contains a multiplexer for selecting between the different sets of instructions of the ALU (e.g., Boolean logic, simple arithmetic, shifter, etc.). This multiplexer is in the critical path and unlikely to be pipelined [Met04], and despite that an FPGA fabric is mainly based on multiplexers, it is poor in implementing wide input multiplexers (see also Sect. 2.6 on Page 104). If carefully applied, runtime reconfiguration allows to integrate more instructions while providing higher performance than a static system. Note that this is in many cases still valid even when considering the configuration overhead. Moreover, partial reconfiguration adds a flexibility to the system that allows to integrate hardware accelerators dynamically to a system like known from the software world.

5.2 Zero Logic Overhead Integration

In this section, we will demonstrate how the Xilinx vendor tools can be used to integrate reconfigurable instructions without any logic overhead. As shown in Fig. 5.3a, we are only interested in binding the signals between the static system (the CPU) and the partial modules (the custom instructions) to a preceisly defined wire of the fabric, called a "PR link", in the following. In order to occupy a wire segment (i.e., use a PR link), we need a path that will use this wire. In other words, there must by somewhere a primitive source (e.g., a LUT output) and another primitive destination (e.g., a LUT input) in our netlist with a requested connection from the source to the destination. However, this creates a path in our netlist but we have still not constrained the routing. This is done by generating blocker macros, that occupy a user specified set of routing resources such that the Xilinx vendor router cannot use these wires for further implementation steps. The blocker concept is introduced in Sect. 3.2.4. Note that we cannot constrain the routing directly in a way that we say "*use wire*$_x$ *for signal*$_y$". We are basically defining a *wire allocation* in a way that we define "*do not use wire*$_z$". However, if we ensure by our allocation, that there is only one possible path remaining, we can actually achieve our goal to bind a signal path to a wire.

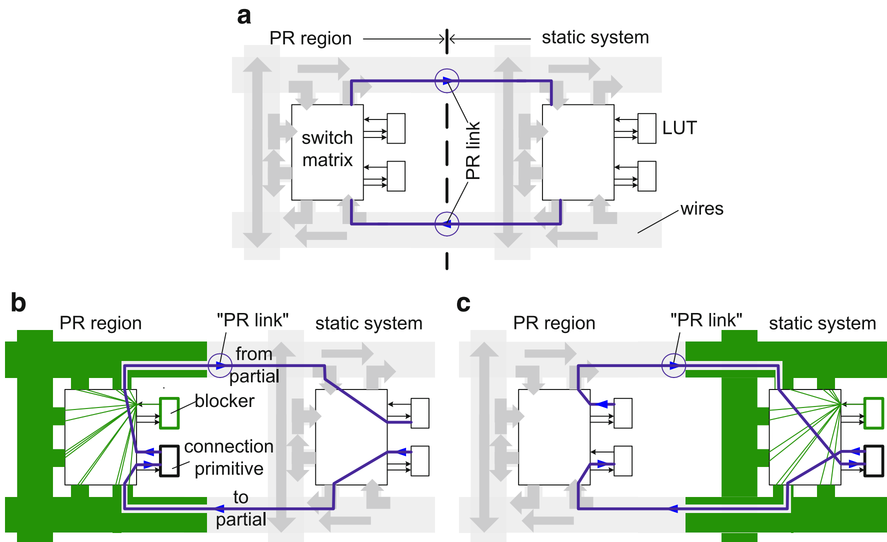

Fig. 5.3 Constraining the routing to use PR links. (**a**) The allocated wires to be used as PR links. These wires carry out the static-partial communication. (**b**) During the implementation of the static system, we place connection primitives into the reconfigurable region that act as a placeholders for the partial modules (here the custom instructions). In addition, we place a blocker *into* the reconfigurable region that conguests all wires except some tunnels for the PR link. (**c**) During the implementation of the partial modules, we place connection primitives into the static region that act as a placeholder fot the static system (here the CPU). In addition, we place a blocker *around* the reconfigurable region that conguests all wires except some tunnels for the PR link

5.2.1 Static System Constraints

With the knowledge of how to create a path and how to constrain this path to certain wire resources of the FPGA fabric, we can implement the static system. The static system contains the CPU and a reconfigurable region. In order to create paths into this region for connecting the operands `OP_A` and `OP_B` (see Fig. 5.1), we place a connection primitive into the reconfigurable region (PR region), as depicted in Fig. 5.3b. This primitive acts as a *placeholder* for the partial module and is the destination for the operand routing. Similarly, for creating a path for the result vector back to the CPU in the static part, we place a placeholder acting as the source for the path. Note that the same LUT primitive (or, to be more preceise, a slice) might be used as a placeholder for multiple input and output signals at the same time. So far, this seams to be pretty much identical to the proxy logic approach. However, we will now add a blocker into the reconfigurable region that blocks all routing resources in this region, except the wires to be used as PR links. Note, that the placement of the placeholders and the blocking is not random and has to support the intended PR link. If we now start the router, we will create the routing of the static system including paths to and from the partial region that are routed using the requested PR links. There are two things to remember: (1) we have not added any logic overhead to the static system, and (2) we only blocked wire resources *inside* the reconfigurable region.

5.2.2 Partial Module Constraints

The partial module implementation (here the custom instructions) is very similar to what we did for the static system. However, all signals directions are now changed and with respect to a custom instruction, the operands are no inputs and the result vector is an output. Consequently, we place a source placeholder as the start for the operands outside the reconfigurable region (i.e., the static region). Respectively, we add also placeholders acting as the destinations for the result vector. Again, placeholders for inputs and outputs can share the same FPGA primitive, as shown in Fig. 5.3. We will now add a blocker around the partial module that conguests all routing resources, except the ones needed to route the operands and results over the PR links. Here it is important, that the blocker releases PR links that are compatible to the PR links used in the static design. Again, there are two things to remember: (1) we have added no logic overhead to the static system, and (2) we only blocked wires *outside* the reconfigurable region. Consequently, when loading a reconfigurable instruction into an reconfigurable island that was created as described for the static system in the last section, there will be no placeholder module visible. The placeholders are only temporarily required to create a path over the PR link.

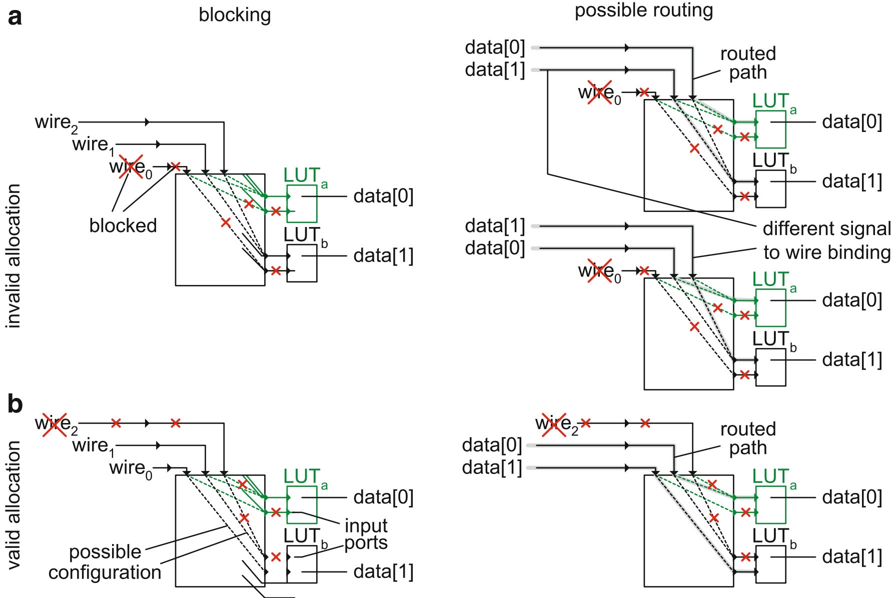

Fig. 5.4 Wire allocation for the zero overhead integration technique. (**a**) Allocating wire_1 and wire_2 results in an unpredictable routing, while (**b**) allocating wire_0 and wire_1 results in an unique routing

5.2.3 Communication Binding by Wire Allocation

The zero logic overhead technique has to follow some rules. Again, by blocking, we can only select the set of wires that are allowed for routing (i.e., wire allocation) but this does not necessary ensure a particular binding of a logical signal to a physical wire. However, the binding is achieved by allocating wires such that *only one unique routing path* can be used to reach the connection macro (see Fig. 5.4). As a consequence, not all wires within a CLB can be used at the same time to implement the routing between the static part and the partial part of a system. This is because in the case that multiple wires are routed from one configurable logic block (CLB) to another, wires must be allocated that cannot be swapped. A possible swapping of wires would allow the router to decide between more than one option for a PR link, which cannot be accepted. A situation of allocating swappable wire resources is shown in Fig. 5.4a. Here, the problem is that both allocated wires can be arbitrary used to connect to both placeholders that used for the data signals `data[0]` and `data[1]`. Consequently, the router has two possibilities to chose from and we cannot gurantee a signal binding to a specific PR link. However, by allocating a different wire set, we leave only one possible path per data signal and we achieve an exact binding to wires, as shown in Fig. 5.4b. Note that designing PR link paths

needs deep knowledge about the FPGA routing fabric including wire resources and possible switch matrix settings. This information is provided by Xilinx individually for each FPGA in a language called XDL [BKT11].

5.3 Implementing Reconfigurable Instructions with the ReCoBus-Builder

The ReCoBus-Builder is originally designed for implementing bus-based systems consisting of many small resource slots that are integrated with the help of macros, as revealed in Sect. 3.2. At this point, we focus only on macros implementing the connection bar architecture (Sect. 2.6.1). For implementing the zero logic overhead approach, we follow the original ReCoBus-Builder flow and perform resource budgeting and define a floorplan that fulfills the resource requirements. Then, we create our communication architecture that will provide connection primitives in the static part of the system as well as in each resource slot. Let us consider the simple case of a connection bar to connect only a single resource slot. We would then basically generate a Xilinx bus macro for an island reconfiguration style. When following the default ReCoBus-Builder flow, we will generate two blocker macros, one for the static design and one for the reconfigurable modules. We will use these blockers for implementing the PR link approach shown in Fig. 5.3. As the blockers generated by the ReCoBus-Builder will not block the wires that are already used for the connection bar macro, the blocker will contain a tunnel for a PR link. The only thing that is now missing are the placeholder primitives. These primitives are taken directly from the generated connection bar macro. Consequently, we can generate compatible placeholder/blocker pairs for both the static system and the partial modules. If we assume a connection bar with one internal wire towards east and another wire towards westwards direction, the resulting primitives and blockers would match the example in Fig. 5.3. The ReCoBus-Builder has a wire database for each supported device. This is used by the tool to check if a wire allocation can ensure PR links without possible swaps as discussed in the last paragraph. With this approach, we can provide four double wire PR links per CLB on a Xilinx Virtex-II FPGA.

As a case study, we consider to integrate up to five different instructions into the system at the same time. Instead of using five individual islands for hosting the instruction modules (as it would be necessary following the Xilinx PR flow), the system uses a more flexible approach with one reconfigurable area that is tiled into five resource slots, as depicted in Fig. 5.5. This has the advantage that modules of different size can be more efficiently integrated into the system by taking a variable amount of slots. The communication architecture has to link the two operands to each slot and the result vector back individually for each slot to an instruction multiplexer. By using different wire resources for the operands and the result vectors that route over different distances, both requirements can

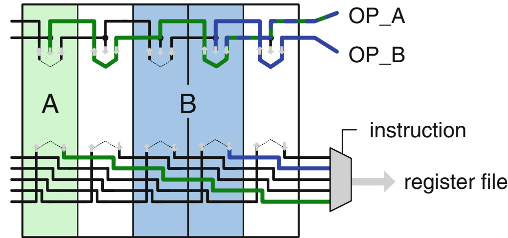

Fig. 5.5 Partial region of the reconfigurable ALU part. The slots can host different sized modules. Due to the interleaved routing of the operands with double lines and due to the available middle access in the middle of a double line, both operators can be accessed in each slot

be properly implemented. By taking advantage of the regular FPGA fabric, the slots can be arranged completely identically, hence allowing free placement of instructions into the reconfigurable ALU. Figure 5.5 reveals a detail of the routing architecture of Xilinx Virtex-II FPGAs that was used to provide slots that are smaller than the routing distance of a wire. In the example, it is assumed that one resource slot is only one CLB wide and that the operands are routed using *double lines* that route two CLBs wide. However, by using a connection in the middle of the wire, which is provided by the routing fabric after a distance of one CLB, and by displacing the start points of the regular routing structure of the two operands by one CLB in horizontal direction, both operands can be accessed in any slot. This is possible by routing the signals in an interleaved manner. Note that it is also possible to route paths by cascading multiple different wires, which would allow to widen the slots (in terms of CLB columns) and to extend the total amount of slots for hosting modules (see Sect. 2.5.2 on Page 81 for more details). The interleaving results in swapping the operands with respect to the placement position (odd or even start slot). However, for instructions that are not commutative, we can use two physical implementations in order to omit the alignment multiplexing. See Sect. 2.5.3 on Page 92 for more details on interleaving.

5.4 Case Study on Custom Instructions

The case study has been implemented with the ReCo-Bus-Builder on a Xilinx Virtex-II XC2V500-5 FPGA. The tool generates regular structured macros together with the surrounding blocker macros that constrain the routing. The implementation follows directly the methodology revealed in Sect. 5.2. The communication macros provide the connection primitives and fix the wire resources. The ReCoBus-Builer generates the all macros (including the blocker) in the Xilinx design language (XDL). While communication macros are instantiated using the HDL flow, the blockers are integrated into the design just before the final route step. A floor-planning view on the system is depicted in Fig. 5.6. The area reserved for hosting

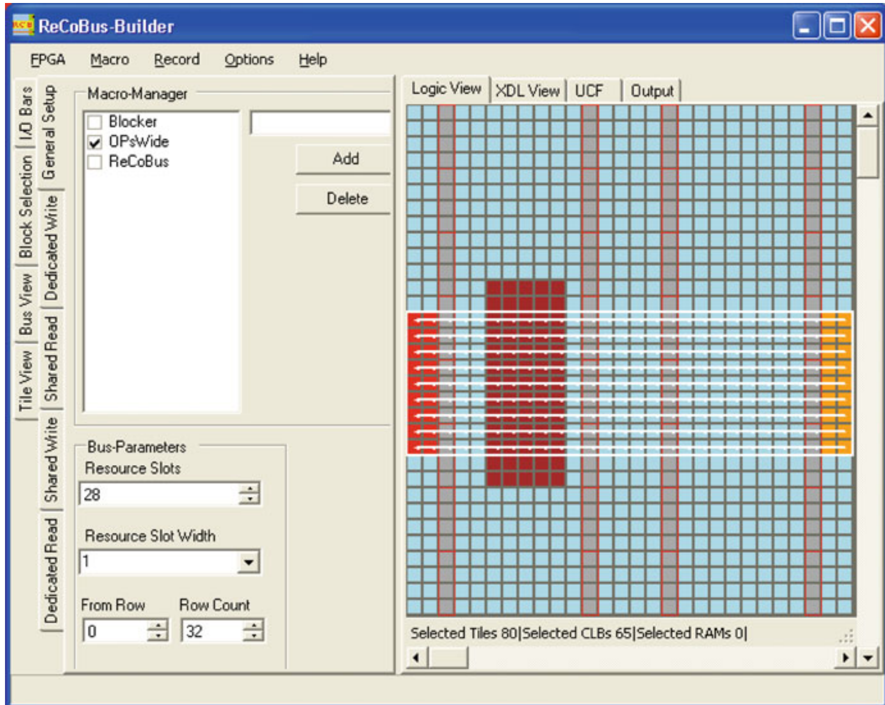

Fig. 5.6 Floorplanning view of the case study. Each gray square represents a CLB that provides eight 4-input LUTs. The five highlighted columns are reserved for hosting up to five different 32-bit instructions

reconfigurable instructions is 8% of the total amount of CLBs that are available on the used device. With five times 48 slices, the PR region provides roughly 15–20% the amount of logic that would be required by an optimized 32 bit soft core processor, such as the Xilinx Microblaze. For the experiments, we used our own MIPS processor implementation that has not been optimized for speed or area, but which can be easily adapted to include reconfigurable instructions.

5.4.1 Static System Implementation

During implementation of the static system, connection primitives that are placed inside the reconfigurable region and that are surrounded with blocker macros have been used to constrain all signals required to integrate the instructions. A screenshot with the static system is shown in Fig. 5.7. The amount of wires that are connected from the static part of the system to the PR region is 2×32 for the operands plus additional eight wires of control signals. In reverse direction, each one out of the five slot delivers a 32 bit result plus additional four flags. This results in a total amount of $64 + 8 + 5 \times (4 + 32) = 252$ wires.

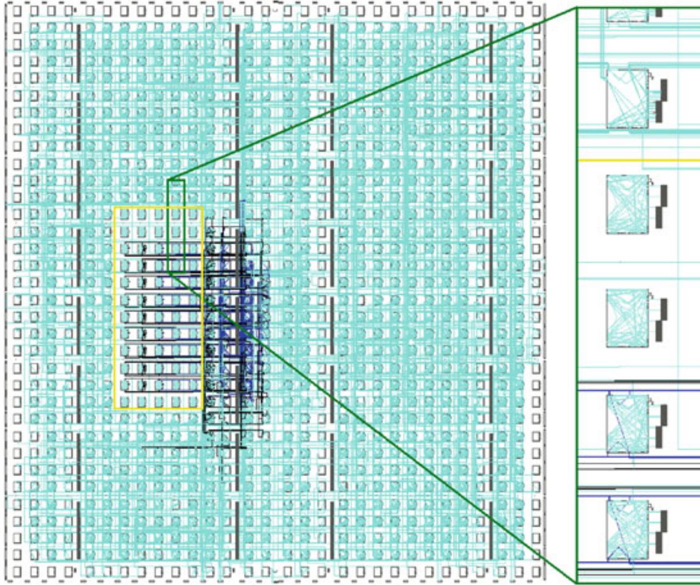

Fig. 5.7 Xilinx FPGA editor view of the static system. Blockers in the PR region prohibit routes of the static system

According to the partial design flow provided by Xilinx, the number of operand bits and control signals has to be multiplied by the number of slots, as that flow does not consider multicast routing to multiple slots without additional connection primitives. Then the slice based macro approach would cost $2\times5\times(72+36) = 1{,}080$ LUTs only for the communication. This is 18% of the available LUTs on the target device and roughly one third of the logic a fully featured 32 bit Microblaze soft core processor would take. Even using the new flow that is based on proxy logic, would still result in a remarkable unnecessary overhead.

When floorplanning a reconfigurable system, it is recommended to consider the underlying FPGA architecture. For example, Xilinx FPGAs are column-wise reconfigured, which should be taken into account by designing the slots vertically. This optimizes the reconfiguration time. A restriction derived from the full column reconfiguration scheme is that no distributed memory can be used directly above or below the PR region as this would corrupt the state of these primitives. Following this rule, partial reconfiguration can be carried out while continuing the system to operate.

FPGAs provide carry chain logic, which are used for different kinds of arithmetic operations. On Xilinx FPGAs, the carry chains include four LUTs per CLB and the chains are arranged in upwards direction. Consequently, we built the system such that exactly two times four operand signal bits and four bits of the result vector are connected in a CLB. Furthermore, the signal vector bits are connected bottom-up (LSBs in the bottom) to follow the carry chain. Without this physical port

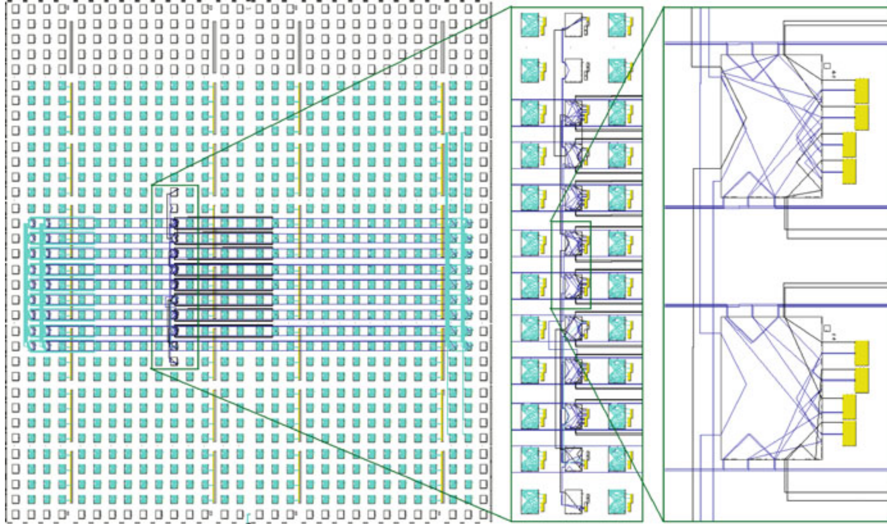

Fig. 5.8 View on the implementation of a CCITT CRC checker instruction

mapping, routing will get very conguested for the modules. In [CPF09], a tool using a simulated annealing heuristic was used to place communication macros around a reconfigurable region that was also used for reconfigurable CPU extensions. Such tools have an excessive runtime as they require a place and route step for each annealing step. It can be assumed, that the final result would be very similar to the here proposed *rule based port mapping* that needs only one place and route run.

5.4.2 Reconfigurable Instructions

For implementing the reconfigurable modules, the complete static system was substituted with a connection bar macro, as depicted in Fig. 5.8. This permits to implement reconfigurable modules in absence of the static system. As can be seen in Fig. 5.8 for a CRC checksum function, a module is surrounded with a blocker macro for restricting modules into strict bounding boxes. This design has no connections to external pins. The timing was constrained with the Xilinx TPSYNC parameter.

5.4.3 Results and Overhead Analysis

Swapping instructions comprises a significant time for writing the corresponding partial bit stream to the right target position. In addition, extra time might be required for computing a placement position or performing some bitstream

Table 5.1 Implementation and performance details

Instruction	# Slices	Slots	Latency (max/av)	b.stream (KB)	t_{conf} (ms)	SW (cycles)	k @ 50 MHz
64-bit XOR	19 (40%)	1	7.04 / 5.95 ns	2.64	0.6	61	492
CCITT CRC	33 (34%)	2*	5.32 / 3.98 ns	5.28	1.2	215/257*	279/233
sat. add/sub	70 (73%)	2	9.89 / 7.81 ns	5.28	1.2	12	5,000
Barrel shifter	90 (94%)	2	11.07 / 7.88 ns	5.28	1.2	143	420
'1' bit counter	214 (89%)	5	11.37 / 8.25 ns	13.2	3	102	1,471
Mask & permute	16 (33%)	1	5.94 / 4.05 ns	2.64	0.6	98	306

*Additional slot to accomplish routing

manipulations. This extra time overhead is implementation dependent and not further considered in the following. However, due to the small size of the systems, most work could be precomputed offline (e.g., a table for the placement position). When taking the decision to use reconfigurable instructions, it is important to know the latency that has to be considered for the reconfiguration process (response time) and the time the processor will require when executing the instructions alternatively as simple software function calls. This determines the breakeven factor k and the system has to trigger a reconfigurable instruction at least k times before gaining a benefit in the total execution time of the system. Note that we use function calls and no traps, as traps are very specific for emulating CPU instructions in software and because traps have a tiny additional overhead that would not occur in case of normal function calls. The configuration times and the execution times for software implementations of the custom instructions (determined in a simulator) are listed in Table 5.1.

The reconfiguration process is relatively slow and would consequently prevent using custom instructions in time critical parts of the software (e.g., interrupts). However this is not problematic as critical software parts should typically not perform complex computations. The breakeven factor k is the number of possible invocations of a particular instruction during the time to configure this instruction. As can be seen, for complex operations, such as the CRC instruction, less than 300 calls of this reconfigurable instruction would pay of the configuration overhead; and even if an instruction can save only a few cycles, this can pay of after just a few thousand cycles. Considering that the saturation addition/subtraction module is used in an image processing application, it can be assumed that it is very likely to trigger this function an sufficient amount of times. It must be mentioned that the listed values are theoretical and the breakeven points will probably be likely higher. This is because the configuration data transfer is in our system in conflict with the CPU (shared memory buses); and even having only a few KB of configuration data results in a burst affecting the CPU. However, reconfigurable instructions are still an interesting option for both saving FPGA resources and gaining performance.

The values in brackets denote the utilization within the occupied slots. Despite that the CRC logic would easily fit into one slot, an additional slot was required to fully route the module. The bitstream size states only the fraction of the partial

module and no static parts. The reconfiguration time is mainly related to the amount of slots that have to be written to the device. A single slot configuration is 11.6 KB on this device which results in 0.6 ms configuration time, when assuming a configuration speed of 20 MB/s. The latency was determined using the FPGA editor. The values are measured between the operand fetching pipeline register through the combinatory path of the instruction and further towards the output of the instruction select multiplexer. The max value denotes the critical path delay and the average delay over all paths.

The examples point out that small FPGA areas are sufficient to include very valuable instructions into a CPU with the help of partial runtime reconfiguration. Despite the small slots, a high number of signals can be interfaced to partial modules.