Efficient Interfacing of Partially Reconfigurable Instruction Set Extensions for Softcore CPUs on FPGAs

Dirk Koch[1], Christian Beckhoff[2], and Jim Torresen[1]

[1]Department of Informatics, University of Oslo, Oslo (Norway)
[2] ReCoBus (Germany),
e-mail: jics@recobus.de

ABSTRACT

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. In this paper, we will sketch that present design techniques include a substantial overhead for integrating reconfigurable parts into the rest of the system. This overhead can cost more logic resources than the actual module implementations. For removing this overhead, we propose a novel technique to constrain the communication resources between the static system and the partial regions. 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 addition, we reveal how such systems can be easily implemented with our tool *ReCoBus-Builder*. Furthermore, we will analyze the overhead in terms of reconfiguration time and present a metric helping to take design decisions.

Index Terms: Partial Reconfiguration, FPGAs, Custom Instructions, CPU Instruction Set

1. INTRODUCTION

With the progress in silicon process technology, FPGA capacity raised by orders of magnitude and more focus in the research on run-time reconfigurable systems has been put on exchanging larger and more complex parts of the system. One of the reasons for this trend is a lack in techniques for integrating reconfigurable modules into a system.

This means that it is not only required to place modules on an FPGA at run-time, it must also be ensured that these modules can communicate with the rest of the system.

This might be communication with other partial modules as well as with the static system. The static system provides the logic that is required at any time, for example, a control CPU, a memory controller, and the interface to the FPGA configuration port.

As running the logic placement and routing is not appropriate to be performed at run-time for any kind of complex module, it is common to provide modules as precompiled bitstreams that can be directly loaded to the FPGA fabric.

While it is possible to manipulate minor parts of the configuration data at run-time (e.g., the placement coordinates [4]), the inner logic and routing of a module will be kept untouched. In order to provide communication to and from a reconfigurable module for a particular signal, the FPGA routing resources have to be constrained such that the same wire resources that cross the partial module border are used for each module at this position.

Let us assume a system with several reconfigurable modules being connected to the backplane bus of a reconfigurable system over time. Then, the module bus signals that have to be routed across the border to the static system are required to use the same wire resources of the fabric among each particular module.

For implementing reconfigurable systems, it requires therefore additional constraints that 1) separate reconfigurable resources from resources that are used by the static system and 2) location constraints on wires that are used to route signals across the border to a reconfigurable region.

Such constraints are vendor tool specific, as HDL languages provide no constraints on routing or logic resources. For example, in VHDL, the only standardized synthesizable attribute is on the encoding of enumerated states, while all other attributes or constraints are not included in the language specification. Originally, FPGA vendors, such as Xilinx Inc., adapted established ASIC design techniques to allow customers to implement circuits on their devices. This

results in limitations when designing run-time reconfigurable systems and some extra effort is required to apply the mentioned additional constraints.

Instead of providing a constraint allowing to bind a signal directly to a definable wire resource, Xilinx proposes to use macros that include routing and that can be placed by the user. By arranging the macro such that one part is located inside the reconfigurable module and another part in the static region of the system, routing can be constrained by internal macro wires. The first macros [10], following this approach, have been based on internal tristate drivers, as depicted in Figure 1.

However, the tristate approach comes along with multiple restrictions, including a relatively low amount of available tristate wires, a risk of interference between multiple reconfigurable modules, and a moderate speed. Moreover, recent FPGAs provide only tristate drivers in their I/O tiles but not inside the fabric.

These issues have been solved with the introduction of slice-based bus macros [7]. As illustrated in Figure 2a), tristate drives have been basically replaced with look-up tables (LUTs). In this approach, the amount of extra logic that is required only for the communication is substantial with two LUTs per bit signal that is crossing the boundary of a reconfigurable module. Note that these are in general the signals in the entity and that a module can easily contain a hundred or more individual signals in its entity. Moreover, 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.

In the latest design flow [11], as released by Xilinx, the slice-based bus macros have been improved by using only one LUT in the partial region per bit signal as depicted in Figure 2b). This technique uses an extra LUT as an anchor (called *proxy logic*) to fix the routing for each bit signal to this look-up table. The routing to the anchors is determined during the implementation of the static system and is then frozen during the physical implementation of each partial module. Note that this requires to reimplement all partial modules in case of a change in the static system. A further difficulty in this design flow is that modules cannot be relocated among different reconfigurable regions even if the regions provide exactly the same shape and resources. This is because the routing to the proxy logic will typically vary among the reconfigurable regions and because the static system may route signals through a reconfigurable region.

As a consequence, modules must include this routing of the static part, hence preventing module relocation. For example, a system with four identical reconfigurable regions for hosting 5 different kind of modules demands 4 * 5=20 individual place and route steps for the module implementation. Again, these steps have to be repeated any time the static system changes. Also the run-time system has to be aware of the different module permutations.

Despite the progress in the Xilinx partial design flow, this flow still comprises a large overhead for the communication and is complex to handle. In the following section, we will reveal a novel approach that does neither demand extra logic for the communication, nor restricting the placement of reconfigurable modules by the static system.

In particular, this permits to migrate modules among multiple different static systems without any additional synthesis or place and route step. The design flow will be demonstrated with the help of a case study in Section 3 where instructions are integrated dynamically into a system using partial reconfiguration. Implementation details are summarized in Section 4 and an overhead analysis will be presented in Section 5.

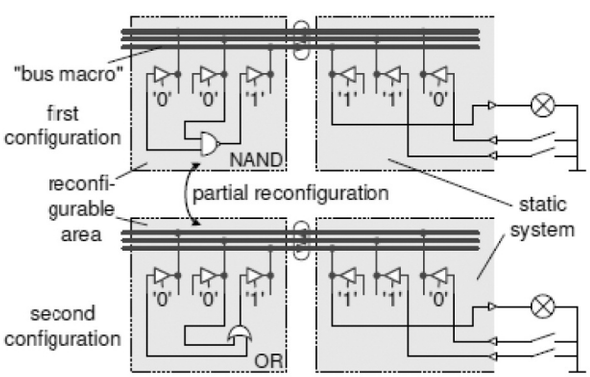

Figure 1. TBUF bus macro [10]. The top and the bot-tom part show two different configurations with the same static system. By fixing the macro placement, signals are bound to tristate wires for crossing the border to the partial region. Such wires have been available in some older FPGAs, (e.g., Virtex-II) and could be linked via buffers located inside the FPGA fabric.

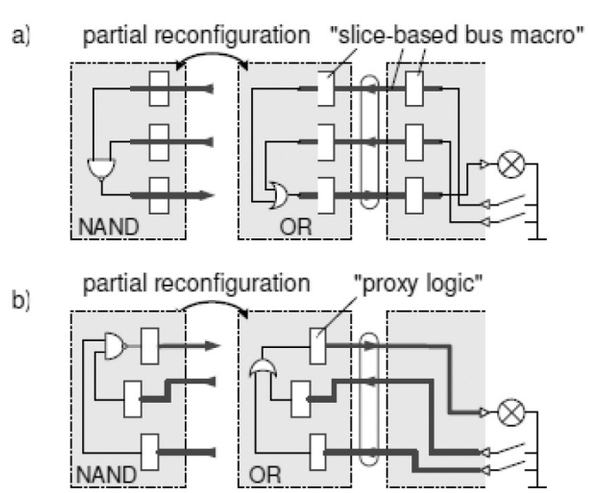

Figure 2. a) Slice-based bus macros [7]. b) Integrating reconfigurable modules with proxy logic [11].

2. ZERO LOGIC OVERHEAD INTEGRATION

As discussed in the last section, one basic problem to be solved in the design of partially reconfigurable systems is to constrain the routing for the interface signals of a partial module during its physical implementation. As illustrated in Figure 3, this is only bound to FPGA routing resources crossing the border to the partially reconfigurable region (PR region), while the rest of the nets are not further restricted to any special routing resources. In the Xilinx vendor tools, there exist no constraints on routing resources that 1) allow binding a signal to a dedicated wire resource and 2) prohibit the usage of certain routing resources. It is possible to define module bounding boxes and prohibit regions, but this is restricted to logic only.

However, it is possible to implement macros that can use any kind of logic or routing resource. Consequently, routing can be restricted with the help of macros. Macros can be instantiated in a design like any other module. During physical implementation, the macros will be placed prior to any other logic. Consequently, the internal logic layout and the routing of a macro will be preserved by the low level tools. In other words, macros can be arranged following a *don't touch* policy. This can be used for 1) binding signals to wire resources and 2) for restricting the routing to selectable resources. The latter one is based on occupying the resources with macros that are intended to be prohibited.

This has been used in the ReCoBus-Builder framework [6] for implementing advanced bus-based reconfigurable systems. In that work, larger reconfigurable regions can be tiled in a relatively fine tile grid. A module can occupy one ore more of such tiles and each tile provides communication to a backplane bus as well as to a circuit switching network for high-throughput streaming data. The communication architecture is provided as large monolithic macros that contain most of the bus logic. These macros are synthesized automatically after floorplanning the system. The macros are regularly designed such that the logic and routing layout is exactly the same throughout all tiles. Consequently, modules can be placed freely within the reconfigurable area, as long as the resources (e.g., dedicated multipliers or memory blocks) match the module requirements. For strictly encapsulating the module implementation from the implementation of the static system, the tool can generate special *blocker macros*. To generate blocker macros, it is possible to define a region in the floorplanner GUI and to select which particular resources (e.g., local wires or longlines) should be blocked. With this information, ReCoBus-Builder generates a corresponding macro that can either be directly instantiated or be transparently included into the design during the place and route phase. The macro generation considers resources that may be already occupied for implementing the communication macros. Consequently, the tool provides a special router for synthesizing blockers as a simple template approach would not guarantee a complete blocking in the case that resources have been allocated for the communication infrastructure.

During the physical implementation of the static system, blocker macros are used to prevent the router to use wire resources within the PR regions. Similarly, during the implementation of the partial modules, blockers ensure that a module does not occupy logic and routing resources outside the specified module bounding box.

By preventing the router to use a selectable set of wire resources, we implicitly force to route signals on the remaining (unblocked) wires. Consequently, we can constrain routing to dedicated resources of the FPGA fabric as sketched in Figure 4. However, in that flow, we still require the shown connection primitives that include an overhead

in terms of area and additional latency for passing the primitives.

In order to avoid the connection primitive, we separate the blocking and the connection primitive into two parts, one containing the macro part of the PR region and another part with the blocker and connection primitive inside the static system. During the physical implementation of one part, the entire other

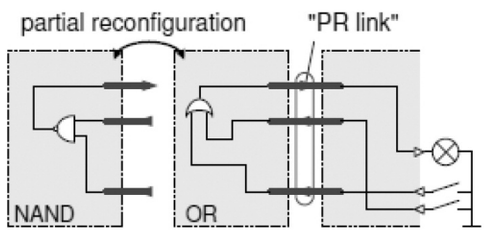

Figure 3. PR link approach. For providing communication with modules located in partial regions, signals will be bound to PR links.].

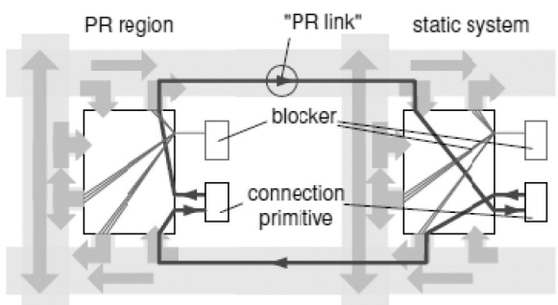

Figure 4. Constraining the routing of signals by blocker macros. The example shows two blocker macros that congest all outgoing signals in the entire tiles such that only the two wires for the PR link remain to be usable by the router. When implementing the static system (partial module), only the left (right) blocker is instantiated for constraining the routing.

one is used for constraining the routing. For example, when implementing a reconfigurable module, we will only instantiate the blocker and the connection primitive in the static part of the system, while not further restricting the PR region. As the only non-blocked resources are the ones assigned as a PR link, all modules will provide an identical interface to the static system. This approach was integrated as a new feature to our tool.

In the case that multiple wires are routed from one configurable logic block (CLB) to another, wires must be allocated that cannot be swapped. Allowing a swapping of wires would allow the router to decide between more than one option for a PR link, which cannot be accepted. Our tool respects this issue by suggesting wires that cannot swap. In the case of Xilinx Virtex-II or Spartan-3 FPGAs, up to four signals can then be bridged per CLB. Thereby it is possible to use the four signals two times when considering both directions. Furthermore, by using wires routing over a longer distance (e.g., double line or hex lines), multiple consecutive CLBs can connect even more PR links per row of CLBs.

Note that the reconfigurable modules and the static system are implemented completely independently from each other. This provides manifold advantages among the Xilinx partial design flow, including module relocation (implying also multiple instances of the same partial configuration bitstream), faster and predictable changes (only changed parts are involved in the physical implementation), and possible IP reuse over multiple different systems. The IP reuse is possible among different devices of the same FPGA family as long as the PR region and the relative position of the PR links are identical.

3. CASE STUDY: RECONFIGURABLE INSTRUCTION SET

A few very promising approaches during the early days in run-time reconfiguration demonstrated successfully the extension of a CPU with customized reconfigurable instructions [9, 1, 3]. For example, the dynamic instruction set computer (DISC) [9], demonstrated a speedup of 80 for a mean convolution application or the GARP, a MIPS processor with reconfigurable instruction set extension [3], achieved a 24 times higher throughput on DES encryption than a SPARC processor and despite that the SPARC operated on a higher clock frequency. Common in all these publications is that relatively small reconfigurable parts demonstrated a material performance improvement. The main reason for the improvement is that major CPU time is spent on relatively small compute kernels, which have been identified by profiling several applications. Accelerating these kernels with optimized instructions can consequently result in a significant performance increase.

Designing a CPU or any other parts of a data path using reconfigurable extensions may in some cases not only speed up a computing intensive kernel, it may also allow a higher clock frequency in general. When assuming the simplified diagram of a CPU datapath in Figure 5a), 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 [8], and despite that an FPGA fabric is mainly based on multiplexers, it is poor in implementing wide input multiplexers, as listed in the following table that denotes the implementation cost of multiplexers on Xilinx Virtex-II / Spartan-3 FPGAs:

Mux inputs	2	3	4	5	6	7	8	9	10	11	12
4-bit LUTs	1	2	2	3	4	4	4	5	5	6	6

As run-time reconfiguration can also be used to multiplex between instructions, the final multiplexer can then be smaller and consequently allowing higher speed of the whole ALU. Of course, this implies a coarse-grained triggering of different kinds of instructions that might be called at a task level, like for example, instructions for either signal processing or cryptography.

The reconfigurable instruction set example is one of the most difficult examples for efficiently using partial run-time reconfiguration. This is because the instructions can be typically implemented with a relatively low amount of logic, while requiring a relatively high number of wires for connecting operands and the result, as illustrated in Figure [5].

Let us assume a 32-bit wide data path as an example, then interfacing a reconfigurable instruction with two operands and one result requires roughly 100 wires. If we further consider two look-up tables to implement the instruction per result bit, then we have more wires to connect than actually LUTs. Note that it can be useful to implement modules without LUTs in some cases, e.g., for bit permutations.

When using the slice-based bus macro approach from Xilinx (see Figure 2a)), the ratio LUTs for interfacing to processing would be in this example: ~2*32+32/2*32=3. Similar worse, the propagation delay would be roughly doubled. The situation will become better when applying the new proxy logic

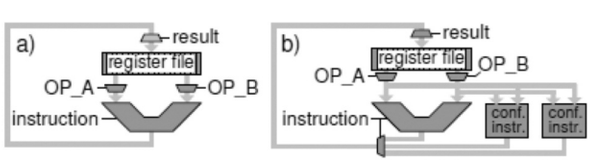

Figure 5. a) Example of a CPU data path. b) Extension of the ALU with multiple reconfigurable instructions. The modules are connected to the operands from the register file and a multiplexer selects between the different results.

approach, but will still end up in an overhead that will prohibit reconfigurable instructions in most practical cases.

The situation changes dramatically when omitting the connection logic for integrating partial modules. Besides reconfigurable instruction set extensions, this is useful for any reconfigurable system, as in many cases I/O bandwidth (which is also related to the propagation delay of the interface) is a performance bottleneck. 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 is common 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 Figure 6. 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 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 6 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. The interleaving results in swapping the operands with respect to the placement position (odd or even start slot). However, even for functions that are not commutative, the same physical implementation may be used at any placement position. This is possible as the connection end port and the middle port of the double lines provide connections to almost the same inputs. There exist also routing wires that route directly between adjacent CLBs, but using these resources would result in a significant higher latency and the total number of this kind of wires is low. Consequently, these wires should be left for the module implementation. More details on cascading and interleaving routing resources can be found in [5].

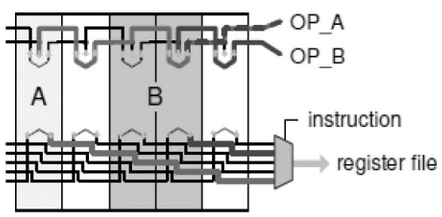

Figure 6. Partial region of the reconfigurable ALU part. The slots can host different sized modules.

4. IMPLEMENTATION AND RESULTS

The case study has been implemented with the tool ReCoBus-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 Section 2. The communication macros provide the connection primitives and fix the wire resources. The ReCoBus-Builder generates all macros (including the blocker) in the Xilinx design language (XDL) [12]. While communication macros are instantiated using the HDL flow, the blockers are integrated into the design just before the final route step.

On Xilinx FPGAs, LUTs are grouped into slices that state the smallest primitive that can be instantiated. Therefore, our tool optimizes slice packing rather than simple LUT packing. A floorplanning view on the system is depicted in Figure 7.

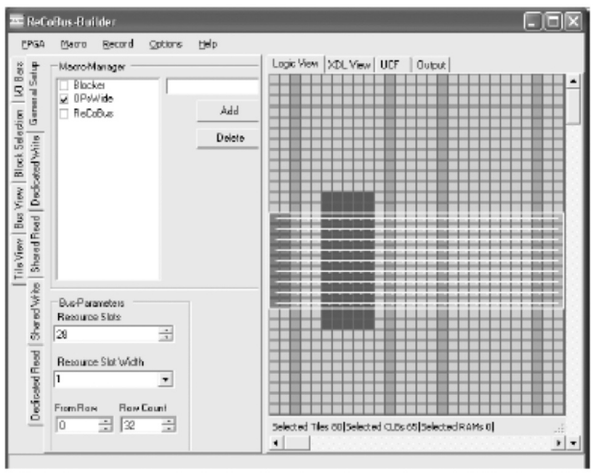

Figure 7. Floorplanning view of the case study. Each gray square repesents a CLB that provides eight 4-input LUTs. The five highlighted columns in the left half of the device are reserved for fosting up to five different instructions. The sreenshot shows also the communication macro linking the operands (32 bit each) and additional control signals. This macro is only rquired for implementing the reconfigurable modules and the blockers.

The area reserved for hosting reconfigurable instructions is 8% of the total amount of CLBs that are available on the used device. With 5 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 processor implementation that has not been optimized for speed or area, but which can be easily adapted to include reconfigurable instructions.

A. STATIC SYSTEM

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 Figure 8. 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 8 wires of control signals. In reverse direction, each one out of the five slots delivers a 32 bit result plus additional four flags. This results in a total amount of 64+8+5*(4+32)=252 wires.

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*5*(72+36)=1080 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 comprises only the half overhead for the proxy logic would still result in a remarkable overhead that can be omitted, when using the here presented approach.

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. The architecture of an FPGA fabric is not completely homogeneous. For example, in Virtex-II FPGAs, carry chains, which are used for all kind of arithmetic operations, are arraigned in upwards direction. This is considered in our tool by allowing to connect bit vectors in the same direction along a resource slot. In [2], 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 may help for very complex systems to improve performance but are assumed to deliver no benefit in the relatively simple designs examined in this paper.

B. RECONFIGURABLE INSTRUCTIONS

For implementing the reconfigurable modules, the complete static system has been substituted with a communication macro, as depicted in Figure 9. This means that the reconfigurable modules can be implemented in absence of the static system. As can be seen in Figure 9 and Figure 10, each module has been surrounded with a dedicated blocker macro for restricting modules into strict bounding boxes. Table 1 lists implementation details of the examined instructions. The values in brackets denote the utilization within the occupied slots. Despite that the

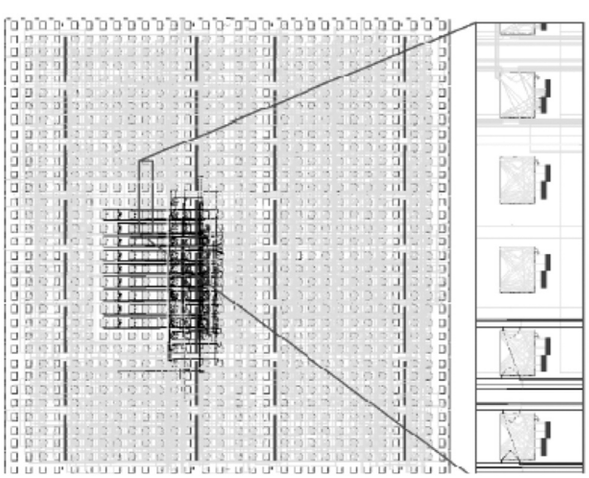

Figure 8. Xilinx FPGA editor view of the static system. The nets for the operands are colored black and the result vectors are blue. Blockers in the PR region prohibit routes of the static system.

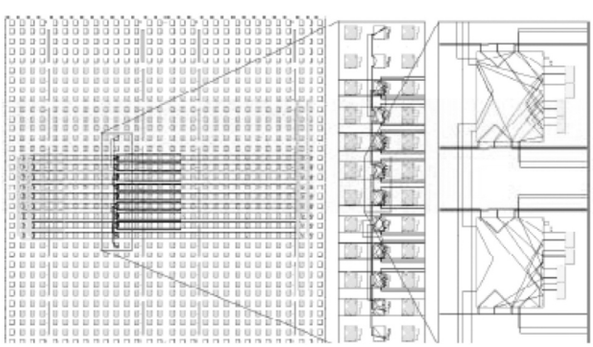

Figure 9. View on the implementation of a CCITT CRC checker instruction.

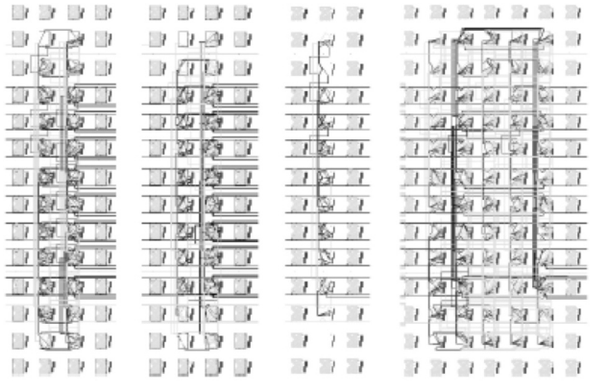

Figure 10. Different reconfigurable instructions: barrel shifter, saturation add/subb, 64 input XOR gate, '1' bit counter (from left to right).

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 manly 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. Note that the control path of the CPU could be easily extended for multicycle operation. Consequently, slow instructions will not decrease performance of the rest of the CPU.

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

5. 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 manipulations. This extra time overhead is implementation dependent and will be not further considered. 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.

Table 1. Implementation details. The relative slice utilization deals with the resources used in the occupied slots.

instruction	slices	slots	latency (max/av)
64-bit XOR	19 (40%)	1	7.04 / 5.95 ns
CCITT CRC	33 (34%)	2*	5.32 / 3.98 ns
sat. add/sub	70 (73%)	2	9.89 / 7.81 ns
barrel shifter	90 (94%)	2	11.07 / 7.88 ns
'1' bit counter	214 (89%)	5	11.37 / 8.25 ns
mask&permute	16 (33%)	1	5.94 / 4.05 ns

(*extra slot for routing)

Table 2. Configuration time and software profiling. The second value for the CRC instruction is for a CPU not featuring a barrel shifter.

instruction	slots	bit-stream [KB]	t conf. [ms]	SW [cy-cles]	k @ 50MHz
64-bit XOR	1	2.64	0.6	61	492
CCITT CRC	2*	5.28	1.2	215/257*	279/233
sat. add/sub	2	5.28	1.2	12	5000
barrel shifter	2	5.28	1.2	143	420
'1' counter	5	13.2	3	102	1471
m & p	1	2.64	0.6	98	306

(*extra slot for routing)

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 2. 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.

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.

6. CONCLUSIONS

In this work, we proposed a novel technique to include reconfigurable modules into a system running on an FPGA without additional overhead for the communication. We demonstrated the technique using a dynamic instruction set case study where multiple modules of different size can be integrated into an ALU by using partial reconfiguration at run-time. Furthermore, we revealed how the approach can be implemented with the tool ReCoBus-Builder. As compared to the competing slice-based bus macro (proxy logic) approach provided by Xilinx, our results demonstrated an area saving of 18% (9%) in the total logic resources available on the used. This was achieved by ommiting extra connection primitives (i.e. look-up tables) in order to integrate reconfigurabe instructions. Furthermore, we analyzed for various custom instructions the corresponding configuration overheads and determined the number of custom instruction invocations required to achieve an overall benefit. It has been proven that with only a relative low number of custom instruction calls partial reconfiguration will pay off and gain a material benefit.

Future work will target on supporting latest devices directly by the tool and on further automating the design process. At present, the trap handler is in charge to manage the reconfiguration of instructions into the system. For future systems, we aim at coupling the reconfiguration with the task handler of the CPU and an autonomous reconfiguration via DMA transfer.

ACKNOWLEDGEMENTS

This work is supported by the Norwegian Research Council founded project Context Switching Reconfigurable Hardware for Communication Systems (COSRECOS) [13], under grant 191156V30.

REFERENCES

[1] P.M. Athanas and H.F. Silverman, “Processor Reconfiguration Through Instruction-Set Metamorphosis: Compiler and Architectures.” *IEEE Computer*, 26(3), 1993, pp 11-18.

[2] J.M. Carver, R.N. Pittman, and A.Forin, “Automatic Bus Macro Placement for Partially Reconfigurable FPGA Designs,” in *Proceeding of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2009, pp 269—272.

[3] J.R. Hauser and J. Wawrzynek, “Garp: a MIPS Processor with a Reconfigurable Coprocessor,” in *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM)*, page12, Washington, DC, USA, 1997, pp 12-21.

[4] H. Kalte, G. Lee, M. Porrmann, and U. Rückert, “REPLICA: A Bitstream Manipulation Filter for Module Relocation in Partial Reconfigurable Systems,” in *Proceedings of the 19th International Parallel and Distributed Processing Symposium - Reconfigurable Architectures Workshop (IPDPS)*, IEEE Computer Society, 2005, pp 151-157.

[5] D. Koch, “Architectures, Methods, and Tools for Distributed Run-time Reconfigurable FPGA-based Systems,” *PhD thesis, University of Erlangen-Nuremberg, Germany*, Erlangen, Dec. 2009.

[6] D. Koch, C. Beckhoff, and J. Teich, “ReCoBus-Builder— a Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for FPGAs,” in *Proceedings of International Conference on Field-Programmable Logic and Applications (FPL 08)*, Heidelberg, Germany, 2008, pp 119-124.

[7] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, “Invited Paper: Enhanced Architecture, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAs,”in *Proceedings of the 16th International Conference on Field Programmable Logic and Application (FPL)*, Aug 2006, pages 1-6.

[8] P. Metzgen, “A High Performance 32-bit ALU for Programmable Logic,” in *Proceedings of the ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays (FPGA*}, 2004, pp 61-70.

[9] M.J. Wirthlin and B.L. Hutchings, “DISC: the Dynamic Instruction Set Computer,” in J.Schewel, editor, *Proceedings on Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing (SPIE) 2607*, Bellingham, WA, 1995, pp 92-103.

[10] Xilinx Inc. “Two Flows for Partial Reconfiguration: Module Based or Difference Based,” May 2002. Available online: http://www.xilinx.com/support/documentation/application_notes/xapp290.pdf Xilinx Inc. “Partial Reconfiguration User Guide,” Rel. 12.1, May. 2010, Available online: http://www.xilinx.com/support /documentation/sw_manuals/xilinx12_1/ug702.pdf

[11] Xilinx Inc. “Partial Reconfiguration User Guide” Dec., 2009. Release 11.4

[12] C. Beckhoff, D. Koch, J. Torresen. “The Xilinx Design Language (XDL) – Tutorial and Use Cases*” in Proceedings of the 6th international workshop on Reconfigurable Communication-centric Systems-on-Chip* (ReCoSoC), Montpellier, France, 2011.

[13] University of Oslo. “Context Switching Reconfigurable Hardware for Communication Systems” project website: http://www.mn.uio.no/ifi/english/research/projects/cosrecos/