High-Level Synthesis of HW Tasks Targeting Run-Time Reconfigurable FPGAs*

Maik Boden[1], Thomas Fiebig[1], Torsten Meißner[1], Steffen Rülke[1], Jürgen Becker[2]

[1]Fraunhofer IIS / EAS Dresden
Zeunerstr. 38, 01069 Dresden, Germany
{boden|ruelke}@eas.iis.fraunhofer.de

[2]Universität Karlsruhe (TH) / ITIV
Engesserstr. 5, 76131 Karlsruhe, Germany
becker@itiv.uni-karlsruhe.de

Abstract

This paper presents a novel High-Level Synthesis (HLS) and optimization approach targeting FPGA architectures that are reconfigurable at run-time. To model a reconfigurable system on a high level of abstraction, we use a hierarchical operation (control and data) flow graph. In order to reduce the overhead for reconfiguring the system, we apply resource sharing to our model to deduce reusable design parts for the implementation. A case study compares our HLS approach with a reference design which was manually coded on Register-Transfer-Level (RTL).

1. Introduction and motivation

Innovative applications in ubiquitous computing (such as mobile embedded and multimedia systems) require a high performance at a reasonable power consumption. State-of-the-art programmable devices are capable to meet this requirements using Run-Time Reconfiguration (RTR) [17]. RTR enables to exchange the functionality of HW partitions while the remaining HW partitions continue processing without interruption.

Several approaches deal with the FPGA-based implementation of RTR systems and the use of high-level design entries [3][15]. Using FPGA architectures, the overhead to reconfigure at run-time affects the system performance significantly. To overcome this limitation, a number of coarse-grained computing architectures implementing RTR has been proposed [8]. According to the higher level of abstraction compared to FPGAs, HLS is a suitable approach for target-mapping and optimization [9][10].

But adaptive computing requires intelligent run-time systems depending on the granularity of the processed application data [1]. Although HLS is sufficient for FPGAs [16], cross-level hierarchical HLS [7] has not been considered for complex RTR systems.

* This work has been partially supported by *SpecVer*, a research project of the Bayerische Forschungsstiftung (www.forschungsstiftung.de)
1-4244-0910-1/07/$20.00 ©2007 IEEE.

Our HLS approach for a system with reconfigurable HW tasks combines both a higher level of abstraction (used by coarse-grained architectures) and a cross-level hierarchical optimization (as proposed in [6]). To model an RTR system target-precisely, we developed a hierarchical operation (control and data) flow graph [4]. In this paper, we introduce an optimization technique based on this model. In order to reduce the RTR overhead we deduce reusable design parts using resource sharing.

This paper is structured as follows: Section 2 introduces design patterns for HLS and an RTR design framework. Special optimization issues targeting FPGAs are covered in Section 3. In Section 4 we evaluate our HLS approach using an manually coded RTL reference design. Finally, Section 5 summarizes the paper and draws out some conclusions.

2. Designing reconfigurable HW tasks

High-Level Synthesis is the mapping of an High-Level Language (HLL) description onto a netlist of design elements and includes optimizations [13]. An HLL notation represents an operation sequence. The design elements are provided by a library and represent RTL counterpart of the HLL operations. HLS comprises three main steps:

- *Allocation* assigns each operation and suitable design elements provided by the RTL library.
- *Binding* maps each design element on a suitable instance onto the target architecture.
- *Scheduling* determines the execution order of an operation sequence according to the operation binding.

This section introduces design patterns and a high-level target-precise model, which we apply for allocating and scheduling the operation sequence. The model represents a hierarchical graph comprising both the description of each HW task and the task schedule of the entire system. For an adequate HLL notation, we developed the Macro Sequence Language (MSL, see Section 4.2 for an example). Binding of operations is discussed in Section 3.

2.1. Design patterns for high-level synthesis

We use control flow templates and operation templates as design elements. Each template represents the RTL counterpart of a HLL construct or operation and provides target specific information (such as required chip area, maximum clock frequency) stored in a library [4].

Control flow templates implement basic constructs of an HLL, such as loops or conditional branches. Thus, a set of these templates can represent the HLL notation of a HW task. We call the control flow templates design patterns. Figure 1 shows the design patterns provided by our design library. We distinguish between the system library (HLL) and the target library (RTL).

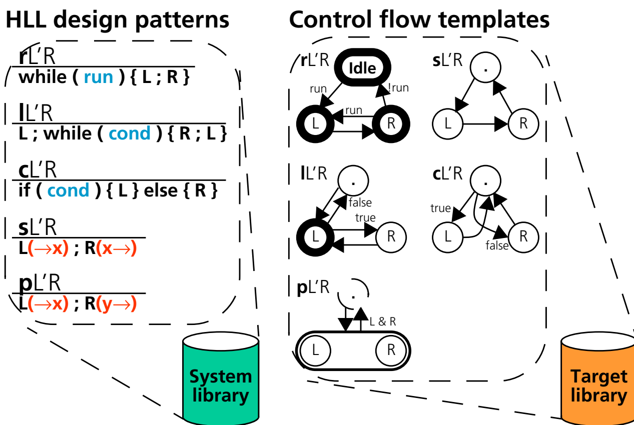


Figure 1. Mapping patterns onto templates

Operation templates are commonly known as design macros. Each macro implements an operation (such as addition or multiplication) respectively a target-specific optimized version (e.g. using HW multipliers).

2.2. High-level target-precise HW task model

We model the operation sequence using a binary tree that represents a hierarchical control and data flow graph, called Binary Macro Tree (BMT). Design patterns are assigned to BMT branches and determine the control flow. Leaves in the BMT represent design macros acording to operations and form the data flow. Thus, executing the operation sequence can be understood as 'walking' through the tree (starting from and ending at the root node).

The *r-pattern* represents the root node of a BMT sub tree which describes the operation sequence of a HW task. Thus, it indicates crossing the level of hierarchy. The BMT root tree marked out by the *r-nodes* comprises the task schedule and global operations (highest level of hierarchy). These operations are excluded from RTR optimization discussed in Section 3.

Figure 2 depicts the mapping of an HLL notation of a HW task onto a sub tree of the BMT of an RTR system.

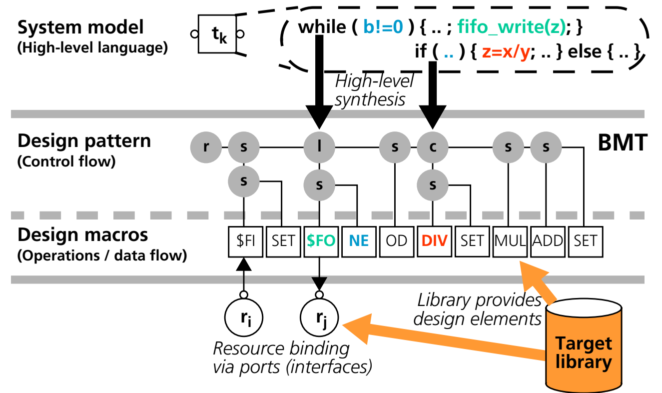


Figure 2. Mapping an HLL notation onto the BMT

For characterizing the RTL implementation of an RTR system we use two techniques of traversing the tree:

- By traversing all nodes (branches and leaves) in order, we can determine the number of instances (patterns and macros), the corresponding chip area, and the critical path which determines the maximum clock frequency.
- By executing the model, we can calculate the task's performance per clock cycle, such as the maximum throughput of computations or the latency of a task in real-time applications.

2.3. Reconfigurable HW task design framework

A reconfigurable system comprises a set of different HW tasks including a task schedule. Its implementation is limited by the available chip area and target-specific resources, such as I/O ports or memories.

The system behavior is characterized by the task, its computation function, and the task schedule. According to Section 2.2, a BMT describes the behavior of a HW task. To define the task schedule, we use a hierarchical BMT. Each sub-tree indicated by the special *r-pattern* represents a HW task. The remaining tree on the top corresponds with the task schedule (see Figure 5).

For mapping the tasks onto the target-architecture, we use a system model consisting of tasks and resources. Tasks perform computations. System resources store data. Tasks and resources can be interlinked via ports to exchange data. A task can operate as a data producer, as a data consumer, or both. In any case, a resource is needed to interlink a producer and a consumer. System resources are also used to interlink tasks with peripherals or I/Os.

Furthermore, a task provides a control port of two signals. The *run-signal* starts processing. The *rdy-signal*

indicates its completion. Thus, the scheduler has to send a token to a task to start it. The other way round, the scheduler will receive a token, if the task has finished.

Figure 3 illustrates a simple system consisting of two tasks. The data-producing task t_p sends data via resource r_{pc} to the data-consuming task t_c. Task t_c consumes data from an input r_i and stores the computation results in the output resource r_o. The tasks scheduler t_s and all system resources r_* are part of the RTR system framework.

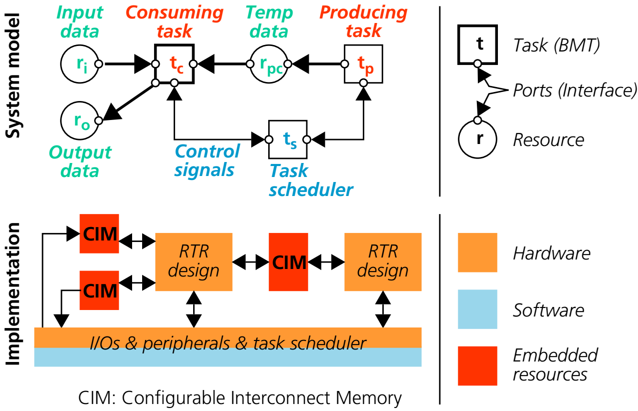


Figure 3. RTR system design example

In our example, Configurable Interconnect Memories (CIMs) implement system resources using multi-port memories. CIMs provide Generic Address Generators (GAGs) for data sequencing [9]. The current realization supports different address patterns and a FIFO mode.

3. RTR optimization targeting FPGAs

RTR causes additional costs (such as configuration memory $c_m(t_i)$ for an RTR chip area $c_a(t_i)$ of a task t_i or the reconfiguration time $c_t(t_i,t_k)$ for switching from task t_i to task t_k). These costs reduce the overall system performance compared to an implementation without using RTR.

Due to the low level of abstraction for describing an implementation, FPGAs provide a very high flexibility for custom realizations of RTR designs. Thus, application-specific operation macros or data path structures could be used to implement HW tasks efficiently. But caused by the fine-grained FPGAs built of Configurable Logic Blocks (CLBs), the RTR costs are to high.

In order to reduce RTR costs, our HLS approach extracts reusable modules. A reusable module combines similar operations used by different tasks. In contrast to processing elements of computing architectures, these modules are application-specific. Thus, an optimized RTR design adapts to the application design and not vice versa. Using a hierarchical BMT enables to apply resource sharing in order to find similar operations.

3.1. Resource sharing in hierarchical BMTs

Resource sharing enables the mapping of two similar operations onto the same computing resource. It reduces the number of required resources by using multiplexers to select the correct operands for computation. Thus, resource sharing reduces the required chip area (Figure 4, top).

In an RTR system, resource sharing is the mapping of similar computing resources onto the same part of the reconfigurable chip area. It enables to exclude this part from the reconfiguration at run-time. Thus, RTR resource sharing reduces configuration overhead (Figure 4, bottom).

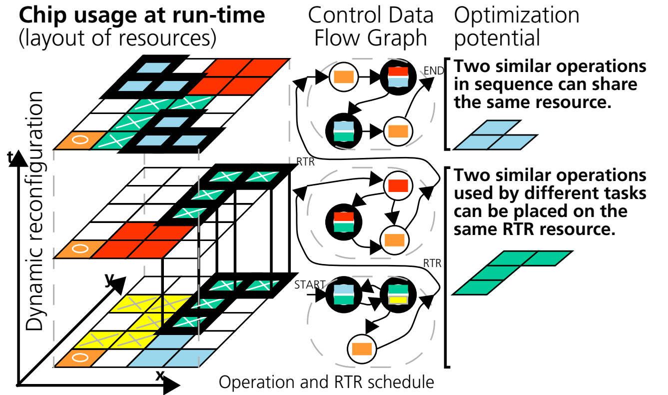


Figure 4. Resource sharing in an RTR system

To identify any possible resource sharing by two similar operations, the operation schedule as well as the RTR schedule have to be considered. Thus, a hierarchical BMT (comprising both schedules) enables to identify any kind of resource sharing including its potential for RTR.

Due to the binary graph representation, resource sharing can be easily determined if the root of the smallest sub tree (containing both operations) represents an *s-pattern*. If the path between the root and both operations contain an *r-pattern*, RTR resource sharing is applicable *(see* Figure 5*)*.

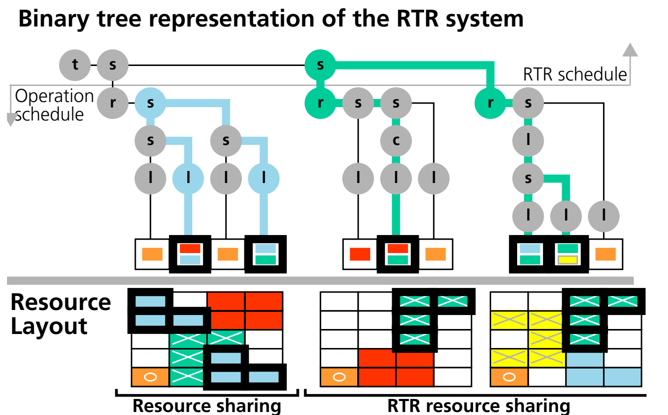


Figure 5. Hierarchical BMT of RTR system example

3.2. Cross-level hierarchical HLS of RTR modules

As depicted in Section 3.1, the implementation of an RTR system depends on the resource layout onto the target architecture. A reusable RTR module has to be laid out on a location which is suitable for all configurations using the embedded resources. In contrast to a shared resource, a reusable RTR module represents a complex component comprising parts of the control and of the data flow.

Extracting a reusable RTR module requires to consider the Control and Data Flow Graph (CDFG) of the operation sequence as well as its mapping onto the synthesized netlist including the operation schedule. This is similar to resource sharing of complex components in hierarchical synthesis as proposed in [6].

For approach we combine a high-level target-precise model (the BMT) [4] with a netlist of generic design elements (the Pattern Macro Netlist, PMN). In contrast to a netlist of logic gates, the PMN distinguishes between control path elements (design patterns) and data path elements (design macros). Thus, a BMT can be mapped onto a PMN and vice versa, which enables the deduction of similar operation sequences for a given RTR module.

Figure 6 shows a system example containing two tasks t_1 and t_2. Each task implements a loop (counting down a pre-initialized index *i*) which calls a function (t_1:*FU1* respectively t_2:*FU2*) that depends on *i*.

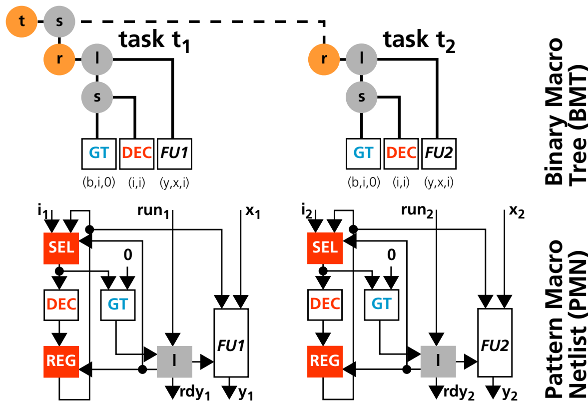


Figure 6. BMT and PMNs of sharing example

Obviously, the *l-pattern* (implementing the control path for a loop) and the down-counter (implemented by using the macros: *SEL*, *DEC*, *REG*, *GT*) could be shared by both tasks. Two additional multiplexers switch the data path according to the active task t_1 or t_2 (see Figure 7, left).

Using RTR, additional multiplexers are not necessary because the environing parts of the design depend on the loaded configuration. Thus, at different points in time also different sources (i.e. t_1 or t_2) can be mapped onto the ports of the RTR module (see Figure 7, right).

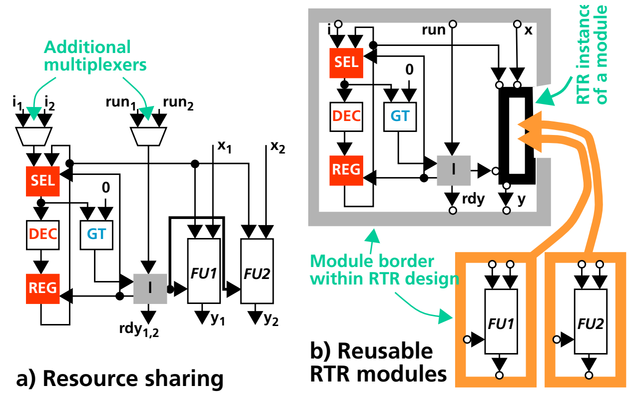


Figure 7. Resource sharing vs. RTR modules

Moreover, using RTR enables to place both functions (t_1:*FU1* respectively t_2:*FU2*) on the same location of the RTR architecture. In consequence, the chip area of the reusable module is similar to the original PMN parts of task t_1 and task t_2.

3.3. Design space using RTR resource sharing

An RTR implementation is characterized by the area costs c_a (such as the number of required CLBs) and the RTR costs c_r (such as number and size of bitstreams).

The initial design is created by mapping of the model neither applying an optimization nor using RTR. Resource sharing reduces the area costs $c_a(sharing) < c_a(initial)$ without raising RTR costs $c_r(sharing) = c_r(initial) = 0$. The area costs c_a can be minimized using RTR, but this causes additional costs $c_r(rtr)$ which affect implementation and performance as well (see Figure 8 and Section 3.1).

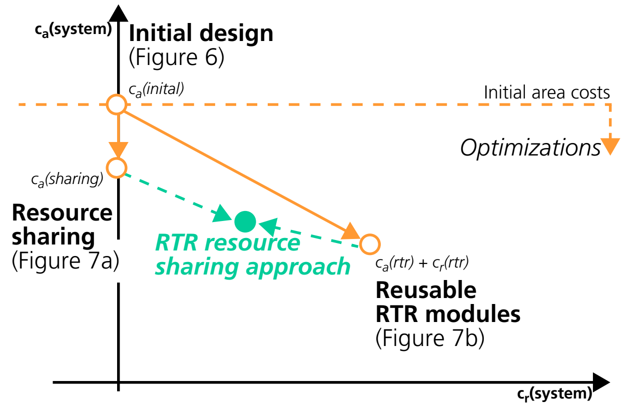


Figure 8. Implementation costs of an RTR system

In practise, reducing area costs to a given limited number of CLBs is sufficient. We realize RTR resource sharing as a combination of both approaches. Thus, RTR is not used if resource sharing already complies the requirements.

4. High-level synthesis vs. RTL design

To evaluate our HLS approach, we implemented a Reed Solomon (RS) codec algorithm as case study. As reference, we used the manually coded and optimized RTR design at RTL of a preceding research project [5]. As target platform for implementation, we choose a Xilinx Virtex FPGA.

4.1. Reed Solomon codec basics and reference

RS codes are block-based channel codes. Typical fields of appliance are data storage (e.g. CD/DVD) and data transmission (e.g. communication networks). Using RS codes allows detecting and correcting of errors within a data stream without retransmissions (called Forward Error Correction, FEC). For this reason redundant information is needed. The approach is to add a checksum of k symbols to n symbols of data. A symbol represents a sequence of m bits. An RS code $RS(l,\ n)$ comprise $l = n + k$ symbols for transmission over the communication channel. An RS code provides an error coverage of $t = {}^{k}/_{2}$ symbols. Due to the principles of Galois Fields $GF(2^m)$ [11] the maximum code word size is limited to $l < 2^m$. Thus, commonly used RS codes based on a $GF(2^8)$ can comprise up to 255 bytes including the checksum.

We selected the $RS(124,\ 128)$ used for applications in context with the high speed communication technology Asynchronous Transfer Mode (ATM) [11]. It transmits 124 data bytes and provides an error coverage of 2 bytes.

The reference design [5] contains six function blocks according to the main tasks of the RS algorithm:

- Task t_1: Encoding including checksum calculation
- Task t_2: Syndrome calculation
- Task t_3: Locator polynomial calculation (Berlekamp-Massey algorithm)
- Task t_4: Error position calculation (Chien search)
- Task t_5: Error value calculation (Fourney algorithm)
- Task t_6: Decoding including error correction

Encoding and Decoding are implemented as a pipelined architecture using building blocks [14].

The design can operate as transmitter (encoder) or receiver (decoder). Concurretly to decoding the received data, syndrome calculation (t_2) is done. If no error was detected, the decoder (t_6) passes the stored data directly to the application. If an error was detected, it is recovered by Berlekamp-Massey algorithm (t_3), Chien search (t_4), and Fourney algorithm (t_5). In this case, t_6 uses the calculated values to correct the error before passing the data to the application.

Figure 9 depicts the function blocks of the reference design including a disturbed communication channel [5]. For simplification, interleaving and deinterleaving are not considered in our case study.

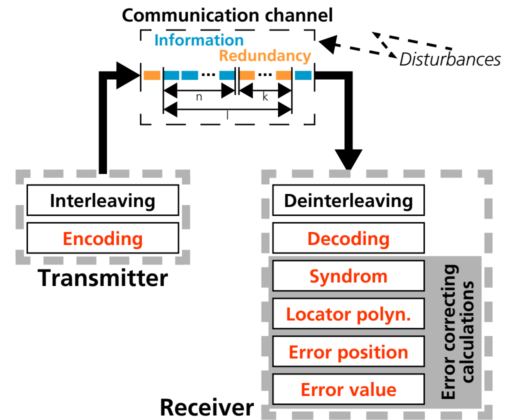


Figure 9. Function blocks of RS reference design

4.2. High-level model and reconfiguration schedule

In order to achieve comparable results, we derived a suitable high-level model from the RTL reference design. Figure 10 depicts the block diagram of tasks ($t_{1\text{-}6}$) and resources ($r_{1\text{-}6}$). Similar to the reference design, we used the task schedule mentioned in the last section. It determines the exclusive operation as transmitter (encoder) or receiver (decoder including error correction).

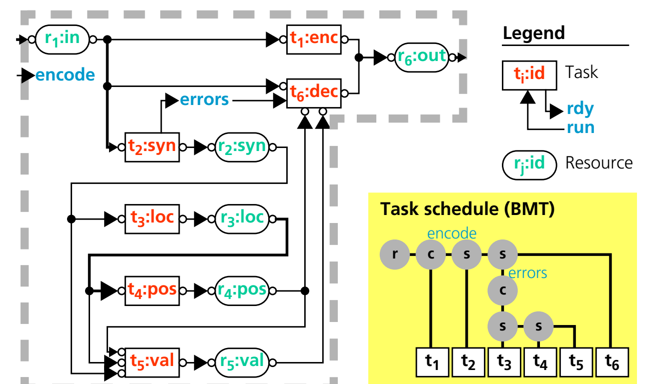
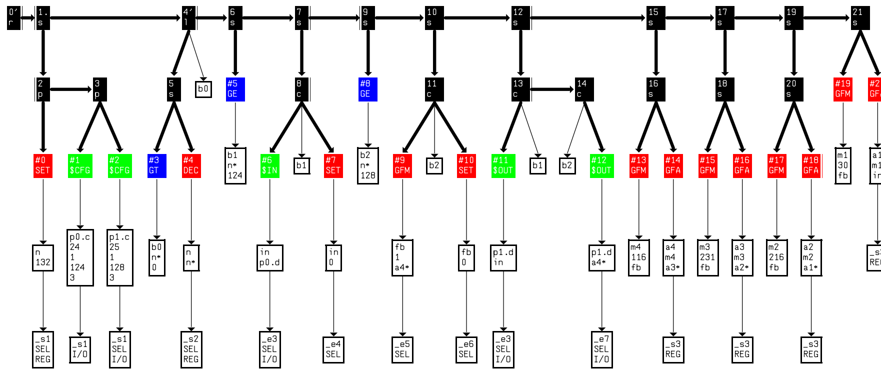


Figure 10. RS tasks, data resources, and schedule

The HW tasks (t_1-t_6) were modelled using MSL (see Section 2). Figure 11 depicts the BMT and the PMN of the encoding task. The corresponding MSL source is given in Figure 12. For visualizing the graphs we use VCG[12].



Pattern Macro Netlist (PMN)

Design patterns (control path)

Signals / Constants (data and control path)

Ports to interconnect memories (data and control path)

Generic macros (data and control path)

Design macros (data path)

Figure 11. RS encode task: high-level model (BMT) and synthesized netlist (PMN)

```
r // Program body (main)
{
}{
SET( n, rs_nk );
$CFG( p0, rf_fi, 1, rs_l, 3 );
$CFG( p1, rf_fo, 1, rs_n, 3 );
l( b0 ) // Main loop (code word)
{
   GT( b0, n, 0 );
   DEC( n, n );
}{
   GE( b1, n, rs_l );
   c( b1 ) // Consume or produce
   {
      $IN( in, p0 );
   }{
      SET( in, 0 );
   }
   GE( b2, n, rs_n );
   c( b2 ) // Feedback to LFSR stages
   {
      GFM( fb, c0, a4 );
   }{
      SET( fb, 0 );
   }
   c( b1 ) // Data or checksum
   {
      $OUT( p1, in );
   }{
      c( b2 )
      {
      }{
         $OUT( p1, a4 );
      }
   }
   // Linear Feedback Shift Register
   GFM( m4, c4, fb );
   GFA( a4, m4, a3 );
   GFM( m3, c3, fb );
   GFA( a3, m3, a2 );
   GFM( m2, c2, fb );
   GFA( a2, m2, a1 );
   GFM( m1, c1, fb );
   GFA( a1, m1, in );
}
}
```

Figure 12. MSL source of RS encode task

4.3. Implementation and test bench

Regarding the RTR technology we used a Xilinx Virtex FPGA as target-platform. Due to the fact that currently the HLS tool suite we developed works with ISE 6.3 only, we choose a XCV1000 device. For logic synthesis we applied Leonardo Spectrum of FPGA Advantage 7.1. The design flow was automated using GNU Make, Tcl/Tk scripts, and Perl scripts to control the synthesis tools.

Figure 13 depicts the test bench and implementation of our case study. Three CIMs (see Section 2.3) are used to enable concurrent register file access by all tasks. This is required by the RS value calculation task t_5.

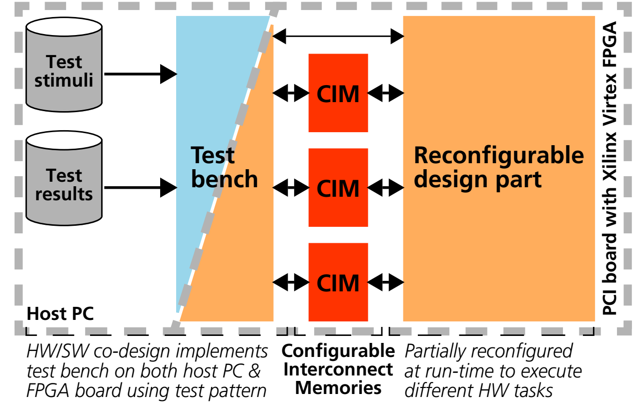


Figure 13. Test bench and implementation

4.4. Implementation costs and performance results

Our evaluation considers the implementation costs as well as the performance results.

Regarding different levels of abstraction, we measured the design effort in Lines of Code (LOC):

- manually coded VHDL source lines at RTL,
- MSL source lines using our HLS approach.

Thus, we are able to compare the synthesis results in relation to the necessary design effort. Table 1 summarizes the ascertained implementation costs of our RS codec.

Table 1. RS codec: implementation costs

	Manual RTL design			High-level synthesis		
	LOC [VHD]	Area [CLB]	Clock [MHz]	LOC [MSL]	Area [CLB]	Clock [MHz]
rs_enc	1,254	263	53.5	69	258	44.1
rs_syn	922	238	44.7	64	224	60.9
rs_loc	1,927	967	35.3	129	767	14.9
rs_pos	1,898	516	28.3	74	356	58.3
rs_val	2,663	733	39.2	82	816	23.3
rs_dec	653	174	64.6	69	248	30.9

Using ModelSim 5.7f for simulation, we measured the performance in clock cycles per task execution. As stimuli we generated pseudo random test patterns providing an uniform error distribution.

Table 2 summarizes the performance results. The typical value represents the average, if a task's run-time depends on the contents of the code word. This applies to task t_3 (Berlekamp-Massey algorithm) and task t_6 (Fourney algorithm). Otherwise we printed the 'typical' value for an *RS(124,128)*.

Table 2. RS codec: performance results

	Manual RTL design			High-level synthesis		
	typ [clks]	**min** [clks]	**max** [clks]	**typ** [clks]	**min** [clks]	**max** [clks]
rs_enc	262	-	-	521	-	-
rs_syn	261	-	-	517	-	-
rs_loc	39	36	40	58	53	61
rs_pos	272	-	-	523	519	525
rs_val	19	-	-	5	-	-
rs_dec	263	-	-	517	-	-

5. Summary and conclusions

Our HLS approach targeting partially reconfigurable FPGAs determines reusable design parts of a RTR system using resource sharing. For this purpose, we introduce a hierarchical operation (control and data) flow graph. This high-level model considers the relevant implementation costs using a target-specific RT level design library.

In a case study we applied our approach to an RS codec application. In comparison to the conventional RT level implementation, we ascertained no general increase of the required area and approximately fifty percent reduction of computing performance. The performance drops down because the current *l-pattern* implementation needs two clock cycles per loop. We will fix this in future.

In contrast to the conventional realization our BMT model enables RTR design optimizations on a high level of abstraction. Thus, next we will add the resource sharing feature to our synthesis framework to explore achievable reductions in the reconfiguration costs.

In future work, we will investigate different methods for resource sharing and design partitioning. Furthermore, we will evaluate other applications of embedded devices, such as multimedia processing or encryption.

6. References

[1] J.Becker, K.Braendle, M.Ullmann, "Reconfigurable Hardware and Intelligent Run-time Systems for Adaptive Computing", Information Technology (it 4/2005), Oldenbourg Verlag, Munich, Germany, 2005

[2] R.A.Bergamaschi, A.Kuehlmann, "A system for production use of high-level synthesis", IEEE Transactions on Very Large Scale Integration Systems, vol.1, no.3, Sep 1993

[3] C.Bobda, M.Majer, A.Ahmadinia, T.Haller, A.Linarth, J.Teich, "The Erlangen slot machine: increasing flexibility in FPGA-based reconfigurable platforms", Int. Conference on Field-Programmable Technology (IEEE ICFPT), Singapore, Dec 2005

[4] M.Boden, S.Rülke, J.Becker, "A High-level Target-precise Model for Designing Reconfigurable HW Tasks", 20th Int. Parallel and Distributed Processing Symposium (IEEE IPDPS), Reconfigurable Architectures Workshop (RAW), Rhodes Island, Greece, Apr 2006

[5] M.Boden, J.Schneider, K.Feske, S.Rülke, "Enhanced Reusability for SoC-based HW/SW Co-Design", 5th Euromicro Conference on Digital System Design (DSD), Dortmund, Germany, Sep 2002

[6] O.Bringmann, W.Rosenstiel ,"Resource sharing in hierarchical synthesis", International Conference on Computer-aided Design (IEEE/ACM ICCAD), San Jose, CA, USA, Nov 1997

[7] O.Bringmann, W.Rosenstiel, "Cross-Level Hierarchical High-Level Synthesis", Int.l Conference on Design, Automation and Test in Europe (DATE), Paris, France, February 1998

[8] R.Hartenstein, "A Decade of Reconfigurable Computing: a Visionary Retrospective", Int. Conference on Design Automation and Test in Europe (DATE), Munich, Germany, Mar 2001

[9] R.Hartenstein, J.Becker, R.Kress, H.Reinig, K.Schmidt, "A Two-Level Hardware/Software Co-Design Framework for Automatic Accelerator Generation", Workshop on Design Methodologies for Microelectronics, Smolenice Castle, Slovakia, Sep 1995

[10] R.Hartenstein, J.Becker, M.Herz, R.Kress, U.Nageldinger, "A Partitioning Programming Environment for a Novel Parallel Architecture", 10th International Parallel Processing Symposium (IEEE IPPS), Honolulu, Hawaii, USA, Apr 1996

[11] A.G.Lomena Moreno, J.C.Lopez Lopez and A.Royo Oreja, "A Pipeline Frequency-Domain Reed-Solomon Decoder for Application in ATM Networks", 14th Conference on Design of Circuits and Integrated Systems (DCIS), Palma de Mallorca, Spain, Nov 1999

[12] G.Sander, Visualization of Compiler Graphs, VCG tool, rw4.cs.uni-sb.de/~sander/html/gsvcg1.html

[13] R.A.Walker, R.Camposano, A Survey of High-Level Synthesis Systems, Kluwer Academic Publishers, Boston/Dordrecht/ London, 1991

[14] W.Wilhelm, "A New Scalable VLSI Architecture for Reed-Solomon Decoders", IEEE Journal of Solid-State Circuits, vol.34, no.3, Mar 1999

[15] Y.Ying, R.Woods, "FPGA-based system-level design framework based on the IRIS synthesis tool and System Generator", Int. Conference on Field-Programmable Technology (IEEE FPT), Hong Kong, Dec 2002

[16] X.-J.Zhang, K.-W.Ng, "An effective high-level synthesis approach for dynamically reconfigurable systems", 4th Int. Conference on High Performance Computing in the Asia-Pacific Region (IEEE HPC-ASIA), Beijing, China, May 2000

[17] Xilinx, Early Access Partial Reconfiguration User Guide For ISE 8.1.01i, UG208 (v1.1), Mar 2006, www.xilinx.com

All registered or unregistered trademarks referenced are the property of their respective owners and no trademark rights to the same is claimed.