Proceedings of the 2010 Forum on specification & Design Languages

Southampton, UK
September 14th-16th, 2010

General Chair:

Prof. Tom Kazmierski
School of Electronics and Computer Science
University of Southampton

FDL is an ecsi event!

FDL 2010

Table of Contents

A Tripartite System Level Design Approach for Design Space Exploration

Peter Brunmayr, Jan Haase, Christoph Grimm
Institute of Computer Technology,
Vienna University of Technology,
1040 Vienna, Austria
{brunmayr, haase, grimm}@ict.tuwien.ac.at

***Abstract*—In this paper a system level design approach is presented, which reduces the effort of integrating low level tools for the evaluation of different solutions during design space exploration. Thereby, low level estimation tools can be utilized for a fast and accurate estimation of the power consumption of different HW/SW architectures.**

The proposed design flow extends the known separation of communication and computation to a tripartite design approach. By separately modeling complex data structures, it is possible to design parts that specify computation directly synthesizable and compilable without major changes. Communication parts and complex data structures are taken from a library or refined manually. Using this approach, the way from a system level model to an actual HW/SW implementation is accelerated and the application of low level power estimation tools becomes possible.

The benefits of this new design approach are demonstrated by the generation of different solutions of a test system of an audio resampler for VoIP systems. Seven different HW/SW solutions are compared concerning their power consumption, latency, and area.

I. Introduction

The partitioning of an embedded system with low to medium quantities corresponds to the task of mapping the functional specification to different integrated circuits (IC) such as FPGAs, general purpose processors and special purpose processors. This design decision is heavily driven by constraints like cost, performance and power and it significantly influences the costs and the optimality of the overall system. The exploration of different partitionings is inevitable to find an optimal realization.

For the classification of different realizations in the design space, accurate estimates of cost, performance and power are required. Although, different solutions for the estimation of design parameters at the system level exist more accurate estimates can be achieved by utilizing low level tools provided directly by the IC suppliers. To use these tools, low level HW and SW implementations are required.

SystemC [1] supports system analysis using simulation and interactive refinement. However, the well established design concept of separating communication and computation [2] used for modeling and simulation does not always provide directly synthesizable and directly compilable computation components. Thus, the analysis of different solutions using low level tools requires manual code conversion. Thereby, the overall effort for generating different HW/SW solutions during design space exploration is increased.

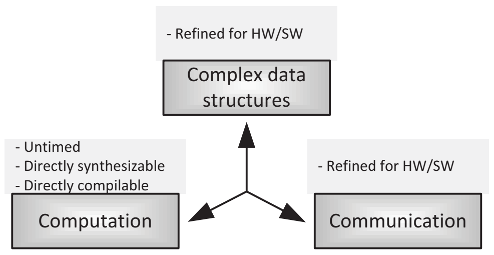

Fig. 1. Separation of communication, data structures and computation.

In this paper, a new system level modeling approach is presented. By separating computation, communication and complex data structures, see Fig. 1, the realization independent design of computation components is enabled. Using high level synthesis (HLS), an untimed system model of computation components can be synthesized together with refined communication components and data structures. Without changing the implementation itself, the same untimed model can be compiled for a software platform using a C++ compiler. Communication components and non-synthesizable constructs such as complex data structures are refined for HW or SW or they are replaced by refined modules from a library.

The rest of this paper is structured as follows: After the presentation of related work in Section II the novel system level modeling approach is introduced in Section III. In Section IV different implementations of an audio resampler test system are presented. Results of these implementations and analysis of cost, performance and power is done in Section V. Finally, Section VI concludes this work.

II. Related Work

One of the first co-design projects were the Ptolemy [3] and the Polis [4] projects, respectively. Both are pioneer works in HW/SW co-design. These academic projects serve as a proof of concept of the general co-design idea and focus mainly on the system level.

Ernst *et al.* [5] present an approach applied in the Cosyma system, where the hardware/software partitioning is based on simulated annealing using estimates of cost and performance. Another system from this period is the Cool system [6]. It facilitates an automatic partitioning approach using performance estimations provided by synthesis tools and compiler. The use of actual synthesis tools and compilers to get performance estimates is similar to our approach, although the system starting with a VHDL description does not fit into a modern embedded systems design flow, where C-based languages like SystemC have emerged for system level descriptions.

An automatic solution published in [7] uses three phases of design space exploration. The general design space is narrowed by using analytical techniques to exclude solutions, which do not fulfill the design constraints. The other two phases use simulation based methods to evaluate different designs. However, for accurate estimates a low level model is required and the simple generation of such a model remains unsolved.

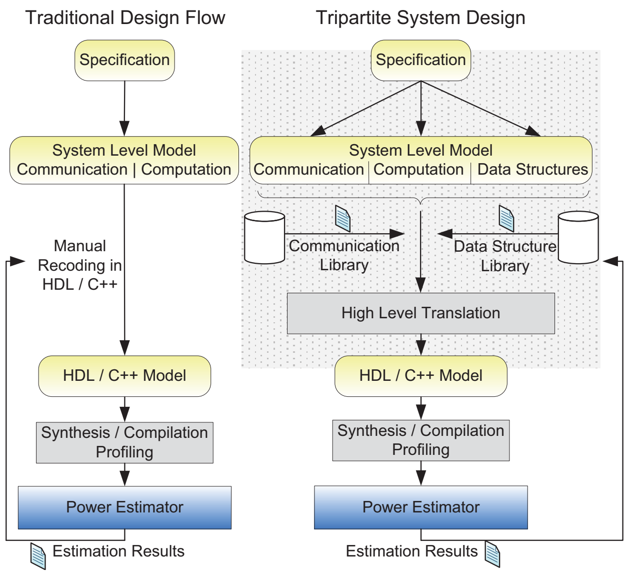

Fig. 2. Design space exploration using the separation of communication, data structures and computation.

III. The Proposed Design Flow

A typical embedded systems design flow is shown in Fig. 2 labeled 'Traditional Design Flow'. After the definition of the specification, a first system level model is designed. At this level, design space exploration is performed, but for the classification of different realizations, estimates of design parameters like power, performance and cost are required. Especially the estimation of power is difficult at the system level. To use more accurate, low level tools provided by the IC suppliers, a low level HW/SW model is necessary, which is typically implemented in C/C++ and a hardware description language (HDL) like VHDL or Verilog.

The traditional design flow requires the manual translation from the system level language to an HDL and C/C++. HLS tools reduce the manual steps, which are necessary to translate a system level model to an HDL model. However if the traditional separation of communication and computation is applied, system models are not both synthesizable and compilable. The reason is that communication components and complex data structures require different code structures for HW and SW.

Therefore, a manual code translation for the generation of an HDL/C++ model is required, if the traditional design flow is applied. Using the new tripartite system design, see Fig. 2, communication, computation and data structures are modeled separately. This enables a synthesizable and compilable model of pure computation. Data structures and communication are refined for HW or SW or taken from a library. The task high level translation in Fig. 2 corresponds to the translation from the system level model to an HDL/C++ model. If a communication and a data structure library are available, this step does not require any manual code rewriting.

The low level HDL/C++ model can either be used directly by a power estimator or it is further synthesized or compiled. Furthermore, to increase the accuracy of the estimation, profiling can be performed at this level. The estimation results are then used at the system level to find the best partitioning of the system.

A. *Tripartite System Design*

Usually the specification is divided into several functional units, which are then implemented at the system level. In this new approach, a functional unit is first separated into functional components of one of the three following types:

- Computation
- Communication
- Complex data structures.

The pure computation is very similar implemented in HDLs compared to software programming languages. Under certain conditions, high level synthesis tools support the direct synthesis of untimed computation code. This enables an almost equal description of computation for HW and SW. Contrary, an accurate description of communication and synchronization behavior needs a lower abstraction level. To define specific timing relations of e.g. interface protocols, a model at the register transfer level (RTL) is necessary. In software, communication is often implemented using facilities provided by the operating system. Hence, operating system (OS) specific code is necessary to realize communication. These differences illustrate the advantage of the separation of communication and computation.

The third type are complex data structures. On the one hand a complex data structure may denote an abstract data type like a stack, a queue, or a list. These data types are in SW usually implemented using dynamic memory allocation, which cannot be synthesized to hardware. An efficient SW implementation of these data structures requires the usage of pointers, which as well is only supported very limited by HLS tools. Following these reasons, these data structures have to be handled separately.

On the other hand complex data structures also denote simple arrays. The decision to model them separately depends on their size. In hardware, e.g. in FPGAs, data structures can be realized as distributed RAM, hence simple registers, or in special memory blocks or even in an off-chip memory. Arrays, which require enough memory, so that the decision to realize them using distributed RAM is not obvious, are classified as complex data structures and are modeled separately. In simple cases, hardware synthesis tools are able to map arrays to distributed RAM or to a RAM block during synthesis. Very often the efficient use of e.g. the multi port capability of such components is only possible via special code constructs or by a direct component instantiation.

After categorizing functional components into the three mentioned types: computation, communication and complex data structures, these types are modeled at the system level. SystemC has been chosen as the system level design language. Fig. 3 shows the process of identifying the different component types in the functional unit and the mapping to SystemC elements. 'Comp' represents a computation component, 'Comm' a communication component and 'Mem' a complex data structure.

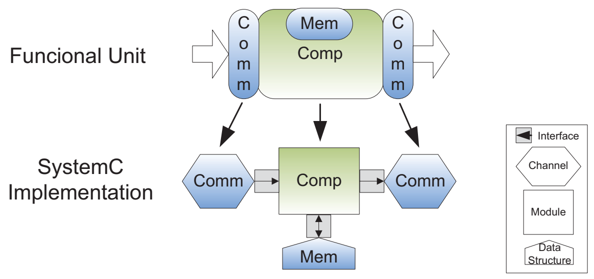

Fig. 3. Identification of functional components and mapping to channels, modules and classes.

a) Interface: With the so-called interface, SystemC provides a capability to separate design components from each other. An interface defines the nature of the data exchange between two components. Each interface specifies a set of methods, which can be used for the data exchange between the components implementing this interface. Through this facility, it is possible to connect different components as long as they have the same interface. This principle is used, when communication components and data structures are replaced by refined components.

b) HW/SW Module: A basic element of SystemC is the module, which is intended to be used for computation purposes. In this work modules are named HW/SW modules. They are connected to channels and data structures via interfaces. To design the modules synthesizable and compilable, the implementation has to be completely untimed. The HW/SW module has active interfaces, which means that data exchange with the module happens via function calls initiated by the module itself. The use of template data types in the HW/SW module and for the interfaces further reduces the difference between a HW and a SW implementation. Thus, data type refinement can easily be done for HW and SW without changing the module itself.

c) Communication Channel: All the communication and synchronization functionality is implemented in channels. The channels have passive interfaces and implement the methods, which are called by the connected HW/SW Module. Four interfaces have been defined to connect channels and modules:

- Serial read interface
- Serial write interface
- Random access read interface
- Random access write interface

The serial interfaces correspond to the simple data exchange, while the random access interfaces are used for write and read operations to and from a specific address.

d) Complex Data Structure: Complex data structures can be used in two different ways. If the data structure is used in between two modules for the communication among these modules, it is consequently modeled in the communication channel. The second possibility is, that the data structure is used by only one module. In this case, the data structure is modeled in a separated C++ class, which is connected to the HW/SW Module using an interface. For basic memory components, like a RAM or a ROM, the random access interfaces can be used. However, for abstract data types more complex interfaces may be required. During the refinement process, the implementation of the data structure is replaced by a refined HW or SW implementation with the same interface.

B. Hardware Implementation

For translating the system model to an HDL model, it has to be refined towards a synthesizable hardware implementation. The implementation of the HW/SW Module does not have to be changed. Only some identifies are replaced by using preprocessor directives. To synthesize a HW/SW Module, the connected communication channels and complex data structures have to be replaced by refined components.

Each channel has to be replaced by a synthesizable channel implemented at RTL. The refined channel defines the cycle accurate input/output behavior of the connected module. Possible channels may implement a simple handshaking protocol, but also complex bus interfaces are possible.

A complex data structure connected to the HW/SW Module, also has to be replaced. Complex data structures like abstract data types are replaced by synthesizable implementations and also simple arrays can be replaced by different implementations. If e.g. a tight timing requires parallel access to the data, the data structure may be realized as a register array. In this case no special code structures are necessary. However, if a special on- or off-chip RAM or ROM block is used, the direct instantiation of this block may be necessary. A memory structure with two ports may be used by two different modules to save resources.

The hardware communication channels, the hardware data structure and the HW/SW Module are synthesized by an HLS tool to RTL modules in Verilog or VHDL. Using the RTL modules, timing information on a clock cycle level is available. The synthesis tools usually also provide estimates of the needed hardware resources. The produced Verilog or VHDL files can be directly synthesized with logic synthesis tools. The results of the logic synthesis can be used by a power estimator to provide accurate estimates of the consumed power.

C. Software Implementation

The system level model can be realized as software without changing the implementation of the HW/SW Module. Using simple preprocessor directives, the SystemC module is transformed to a C++ class. The method which implements the actual computation of the HW/SW Module, can be executed as a thread or a task in an OS. To enable the connection of HW/SW Modules with refined channels and data structures, a basic version of the interface concept used in SystemC has been reproduced for the software implementation.

For software implementation, communication channels and complex data structures have to be replaced by refined components. Pure software channels usually handle internally the communication and synchronization between the threads/tasks using OS facilities like semaphores or mailboxes. Dynamic data structures may be replaced by efficient implementations with static memory management. Depending on the used platform, the data types of the HW/SW module can be adapted. More complex platforms may also accept floating point data types, while many components support only fixed point data types.

The resulting implementation is a pure C++ software implementation and can be integrated into the OS of the used platform. By simulating, emulating or testing the implementation on a prototyping board, the functionality can be verified and the performance can be measured. These results help to accurately estimate the utilization of different parts of the processor. Using low level power estimation tools this information can be used to accurately estimate the consumed power. Furthermore, the estimation results can be used at the system level to find the best partitioning.

IV. Case Study: Audio Resampler

One example which has been designed using this new design flow is an audio resampler for embedded voice over IP (VoIP) systems. Since the sender and the receiver are different systems at different locations, the desired sampling rate is generated from different clock sources. This leads to slightly different sampling rates. If no conversion is performed either one sample is at least taken twice or one or more samples are lost. This leads to a periodic disturbing noise during the VoIP call. One solution is the recovery of the sender's clock at the receiver, followed by a resampler [8], which converts the audio signal from one clock domain to the other. Such a resampler has been designed and implemented in HW and in SW.

The resampler, inspired by [9], basically implements Eq. 1. This equation denotes the process of ideally reconstructing the corresponding bandlimited analog signal of the digital input signal x, which has been sampled with the transmitter's sampling rate $\frac{1}{T_s}$. The continuous signal is then sampled with the receiver's sampling rate $\frac{1}{T'_s}$, m ranges over the integers, h_s being a $\frac{sin(x)}{x}$ function,

$$x(mT'_s) = \sum_{n=-\infty}^{\infty} x(nT_s)h_s(mT'_s - nT_s). \qquad (1)$$

Eq. 1 denotes an ideal low pass filter. For the implementation an approximation with a finite impulse response h_s has been designed. The actual coefficients for each output value depend on the current phase difference $mT'_s - nT_s$ of the two clock domains. To ensure accurate coefficients, an oversampled version of the impulse response has to be stored.

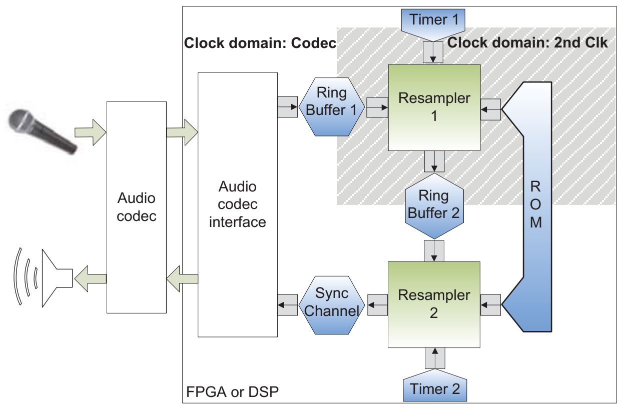

Fig. 4. Test architecture for the audio resampler with a second clock domain.

A simple system to test the resampler and to apply design space exploration by evaluating different solutions is shown in Fig. 4. The system basically converts a recorded audio signal to a second clock domain using a resampler. A second resampler is used to convert the signal back to the original clock domain. Analog to digital and digital to analog conversion is handled by an audio codec chip, which represents the first clock domain of the system. The second clock domain is generated in HW or SW. If the resampling works correctly, the output signal corresponds to the input signal, without any errors. Only a small noise is added during the resample process.

A. Tripartite System Modeling

The whole system is modeled using the tripartite system design methodology. The resampler is separated into data structures, communication and computation components. Each resampler always needs the current plus several older input values to calculate one resampled output value. Therefore, a ring buffer is used to store the input values. With every rising edge of the input clock a new input value is written to the ring buffer. The ring buffer and the synchronization to the input clock signal is identified as a communication component and is moved to the RingBuffer channel. This channel is used at the inputs of both resamplers.

The impulse response of the filter is a large data structure. Therefore, it is modeled in a separated class (ROM), which is connected to the resampler module using the random access read interface. To load the correct coefficients the current phase difference between the clock domains is needed. Such timing information is neither really communication/synchronization nor pure computation. Obviously, it is realization dependent and cannot be realized in an untimed HW/SW module. Thus, it is also implemented in a separated class. These classes (Timer 1, Timer 2) are connected to the resampler modules using the serial read interface.

The actual resample process, the calculation of a new output value is realized in a HW/SW module. This module is design completely untimed. The synchronization to the output clock domain is managed in the channel connected to the resamplers output. The synchronization of the first resampler is done in RingBuffer 2. The second resampler is connected to a SyncChannel. This is a simple serial channel, which blocks until the next rising edge of the synchronization signal, which is in this case the clock of the codec.

The first RingBuffer and the SyncChannel are connected to an audio codec interface module, which configures the codec and handles the communication with it.

B. Design Space Exploration

The designed system model has been used to generate seven different HW/SW solutions. All channels and data structures are replaced by refined HW or SW components, while the resampler module can be used completely unchanged. One solution realizes the system in software, which targets a TMS320C6455 DSP from Texas Instruments Inc. [10] running at 1.2 GHz. In this case, the audio codec is connected to the multi channel buffered serial port (McBSP). The two clock signals are the McBSP receive interrupt (Clock domain: Codec) and a hardware timer interrupt (Clock domain: 2nd Clk). Another hardware timer is used as a clock cycle counter, read by the timer modules. The two HW/SW modules and the audio codec interface run as parallel tasks. In software the coefficients are stored in a simple constant array and the synchronization between the clock signals and the tasks is implemented using semaphores.

After replacing channels and data structures with refined hardware components, six different hardware solutions are generated using the HLS tool Cynthesizer from Forte Design Systems [11]. These solutions are targeting a realization on an Xilinx XC4VFX20 Virtex4 FPGA running with 100 MHz [12]. The basic difference of these implementations is the used memory for the ring buffer, which can either be a dual port block RAM or distributed RAM. Other differences arise from applying different synthesis constraints. In all realizations, the impulse response is implemented as dual port block ROM, where each resampler is connected to one of the ports. The Timer channels are implemented as clock cycle counters.

All seven HW/SW solutions have been analyzed by power estimators. The results of this analyses are presented in Sec. V. Furthermore, all generated realizations have been tested on prototyping boards, the hardware solutions on a ML405 from Xilinx and the software solution on the DSK6455 from Digital Spectrum [13]. Fig. 5 shows measurement results of the resampler implementation on the prototyping boards. A Sine signal with a frequency of 1020 Hz has been generated on a PC and sent to the prototyping board. The spectrum with the lower noise floor corresponds to a measurement of the test setup itself, where the FPGA/DSP sent the test signal unaltered back to the PC. The second measurement included the resampler test system used to convert the signal to a second clock domain and back to the original domain. In this case, the overall noise floor is increased by 5 (FPGA) to 10 dB (DSP). Both implementations perform a smooth conversion and prohibit the generation of a disturbing periodic noise.

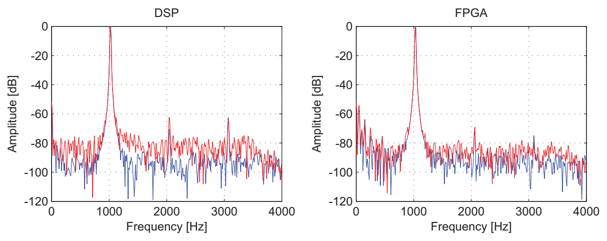

Fig. 5. Measurement results of FPGA and DSP implementation.

V. Results

The resampler has been designed for a sampling rate of 8 kHz, which results in a cycle duration of 125 μs. This timespan is the upper limit the calculation of a new resampled output value may take. The software solution realized with a DSP with 1,2 GHz requires 5450 cycles for this calculation, which corresponds to a calculation time of $4,54$ μs. Obviously, the DSP is oversized for this application and a smaller, slower chip would satisfy this applications demands.

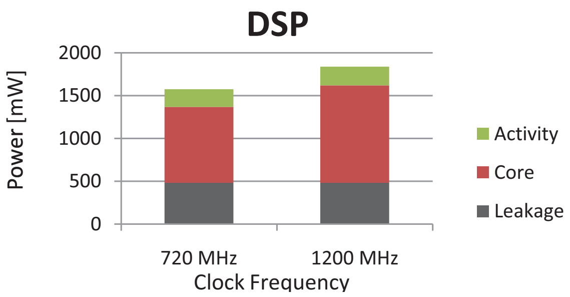

Fig. 6. Power consumption of DSP solution.

For estimating the power TI provides a spreadsheet based power estimator [14]. Fig. 6 shows the estimated power consumption of the resampler test system for two different clock frequencies. With a frequency of 1.2 GHz a power consumption of around 1800 mW is estimated. This value is composed of three components. The power consumed by leakage currents is the second largest component, see Fig. 6. It depends solely on the IC type and the environment temperature. For a typical room temperature of $25°C$, the used TMS320C6455 has a leakage power of around 480 mW.

The largest component is the core power, which is composed of static power and power consumed by the phase-locked loop and clock tree. It mainly depends on the used frequency. By reducing the clock frequency to 720 MHz, which is the minimum frequency of the TMS320C6455, the overall power consumption can be reduced to around 1600 mW.

The third component is the dynamic power, see 'Activity' in Fig. 6. It is the smallest component and the only one, which depends on the application code itself. The used design flow enables profiling by simulation or even emulation. Thereby, the CPU-load has been measured, which has been used to estimate the CPU and peripheral utilization. The power estimator avails this information to calculate a more accurate estimated power consumption.

Since, the used DSP and the frequency are oversized for this application, a logical step would be the switch to a smaller, slower DSP to save cost and energy. Due to the used design flow, most of the code can be reused, as long as a C++ compiler exists for the new processor. Only channels and data structures may have to be changed, if they include processor or OS specific function calls.

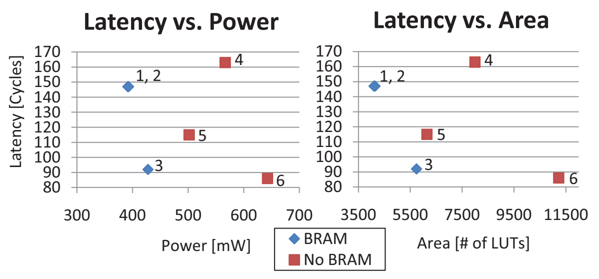

Fig. 7. Latency vs. Power/Area of FPGA solutions.

Another alternative is the realization in hardware on an FPGA. Using HLS six different solutions have been generated. Their latency, the estimated area and the estimated power consumption is shown in Fig. 7. Latency and area information is provided by the synthesis tool. The power estimation has been performed using XPower Analyzer from Xilinx. It uses the post place and route model of a design for the estimation. At this point in the hardware design flow, information about the exact implementation in the FPGA is available. This enables a very accurate estimation of the required power. To further increase the accuracy, it is possible to apply profiling by simulating the post place and route model with application typical input signals.

The hardware architecture generated without any specific constraints, see 1 in Fig. 7, has a latency of 147 cycles. With a clock frequency of 100 MHz this leads to a calculation time of $1,47\ \mu s$. Without specific constraints, the design is optimized for area, which results also in the lowest power consumption. By applying latency constraints two other solutions are generated, see 2 and 3 in Fig. 7. The design parameters of solution 1 and 2 are almost equal. In solution 2, the number of needed look-up-tables (LUT) is slightly reduced by using two multipliers in parallel. However, the required latency cannot be reduced. A significant reduction of the latency to 92 cycles can be attained by solution 3. This can be achieved by parallelization, which results in an increased area and power usage.

Three other realizations (4, 5, 6) are generated by replacing the ring buffer channel. The original channel uses block RAMs, which allow only sequential access to the memory. By using a ring buffer with distributed RAM, the fastest solution, see 6 in Fig. 7 can be generated. It needs only 86 cycles, but consumes much more area and power.

Obviously, a trade off between used hardware resources, power and the needed latency has to be found. The decision for the best solution depends on the given design constraints like minimum performance and maximum power and cost.

VI. Conclusion

The novel tripartite system design approach has been presented. The segmentation into communication, computation and complex data structures at the system level can be exploited to design computation modules directly synthesizable and compilable. This enables the simple generation of different HW/SW implementations. Thereby, the efficient use of low level power estimation tools for design space exploration becomes possible. Obviously, the presented example is only a test system, but it presents the simplified exploration of the HW/SW design space.

References

[1] SystemC^{TM}. [Online]. Available: http://www.systemc.org

[2] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System level design: Orthogonalization of concerns and platform-based design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, pp. 1523–1543, 2000.

[3] A. Kalavade and E. A. Lee, "A hardware-software codesign methodology for DSP applications," *IEEE Design and Test of Computers*, vol. 10, no. 3, pp. 16–28, September 1993.

[4] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara, *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. The Kluwer Academic Press, 1997.

[5] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," *IEEE Design and Test of Computers*, vol. 10, no. 4, pp. 64 – 75, 1993.

[6] R. Niemann and P. Marwedel, "An algorithm for hardware/software partitioning using mixed integer linear programming," in *Proceedings of the Euopean Design and Test Conference*, 1996.

[7] S. Mahnty, V. K. Prasanna, S. Neema, and J. Davis, "Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation," in *Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems*, 2002.

[8] P. Brunmayr, H. Wohlmuth, and J. Haase, "An efficient FPGA implementation of an arbitrary sampling rate converter for VoIP," in *Austrochip 2009*, October 2009, pp. 33–38.

[9] J. Smith and P. Gosset, "A flexible sampling-rate conversion method." in *IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 2, San Diego, March 1984, pp. 19.4.1 – 19.4.2.

[10] TMS320C6455 Data Sheet. Texas Instruments. [Online]. Available: http://www.ti.com/lit/gpn/tms320c6455

[11] Cynthesizer Data Sheet. ForteDS. [Online]. Available: http://www.forteds.com/products/cynthesizer_datasheet_2008.pdf

[12] Xilinx. Virtex-4 Family Overview. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf

[13] TMS320C6455 DSK Tech. Ref. [Online]. Available: http://c6000.spectrumdigital.com/dsk6455/v2/files/6455_dsk_techref.pdf

[14] Power Spreadsheet for TMS320C6455. [Online]. Available: http://focus.ti.com.cn/cn/lit/an/spraae8b/spraae8b.pdf