In Proceedings of the Wireless Innovation Conference and Product Exposition, Washington DC, USA, November 2010.

A Lightweight Dataflow Approach for Design and Implementation of SDR Systems

Chung-Ching Shen, William Plishker, Hsiang-Huang Wu, and Shuvra S. Bhattacharyya
Department of Electrical & Computer Engineering
and Institute for Advanced Computer Studies
University of Maryland, College Park, Maryland, USA
{ccshen, plishker, hhwu, ssb}@umd.edu

Abstract

Model-based design methods based on dataflow *models of computation are attractive for design and implementation of wireless communication systems because of their intuitive correspondence to communication system block diagrams, and the formal structure that is exposed through formal dataflow representations (e.g., see [2]).*

In this paper, we introduce a novel lightweight dataflow (LWDF) programming model for model-based design and implementation of wireless communication and software-defined radio systems. The approach is suitable for improving the productivity of the design process; the agility with which designs can be retargeted across different platforms; and the quality of derived implementations. By "lightweight", we meant that the programming model is designed to be minimally intrusive on existing design processes, and require minimal dependence on specialized tools or libraries. This allows designers to integrate and experiment with dataflow modeling approaches relatively quickly and flexibly into existing design methodologies and processes.

1 Introduction

Software defined radio (SDR) attempts to implement most of the complex signal handling involved in receivers and transmitters of radio using software, rather than being based on special-purpose hardware. For example, modulation and demodulation and all of the signaling protocols of a radio are implemented as software functions [1]. Such designs should be able to rapidly target a variety of platforms, such as programmable digital signal processors, field programmable gate arrays (FPGAs), and graphics processing units for functional prototyping and validation.

Model-based design has been explored widely over the years in many domains of embedded processing. In model-based design, high level application subsystems are specified in terms of functional components that interact through formal models of computation (e.g., see [7]). By exposing and exploiting high level application structure that is often difficult or impossible to extract from platform-based design tools such as SPEX [12] for SDR applications, model-based approaches facilitate systematic integration, analysis, synthesis, and optimization that can be used to exploit platform-based tools and devices more effectively.

Dataflow is a well known computational model and is widely used for expressing the functionality of digital signal processing (DSP) applications, such as audio, digital communications, and video data stream processing (e.g., see [4], [11]).

In dataflow models of computation, applications are modeled as directed graphs, where vertices (*actors*) represent computational functions and edges represent inter-actor communication channels. Actors produce and consume data from edges, and edges buffer data in a first-in first-out (FIFO) fashion. Dataflow actors can represent computations of arbitrary complexity as long as the interfaces of the computations conform to dataflow semantics, which means that actors produce and consume data from their input and output edges, respectively, and each dataflow actor executes as a sequence of discrete units of computation, called *firings*, where each firing depends on some well-defined amount of data from the input edges of the associated actor.

When designing SDR applications using dataflow techniques, retargetability is an important issue since such a wide variety of platforms is available for targeting SDR implementations under different kinds of constraints and requirements. Following structured design

methods, designers should be able to port implementations of dataflow graph components (actors and FIFOs) across different platform-oriented languages, such as C, C++, SystemC, Verilog or VHDL. Furthermore, by promoting formally specified component interfaces (in terms of dataflow semantics) and modular design principles, dataflow techniques provide natural connections to powerful unit testing methodologies, as well as automated, unit-testing-driven correspondence checking between different implementations of the same component (e.g., see [16])).

In this paper, we introduce a novel dataflow-based design approach for model-based design and implementation of wireless communication and software-defined radio systems. In contrast to the framework in [8], which focuses on a specific actor language (C++), specific application domain (speech recognition), and specialized dataflow model of computation (synchronous dataflow [11]), our proposed lightweight dataflow approach facilitates the processes of cross-language, cross-platform migration and prototyping on a variety of signal processing application domains, including SDR, and efficient implementation by providing a structured, easily-retargetable approach for implementing dataflow based DSP systems. Our approach also allows designers to experiment with different dataflow modeling approaches such as SDF [11], CSDF [5], or CFDF [14].

2 Background

Synchronous dataflow (SDF) [11] is a restricted form of dataflow that is useful for an important class of DSP applications, and is used in a variety of commercial design tools. SDF in its pure form can only represent applications in which actors produce and consume constant amounts of data with respect to their input and output ports — that is, data-dependent or otherwise dynamic data rates are not supported in SDF. By using SDF, we can efficiently check conditions such as whether a given SDF graph deadlocks, and whether it can be implemented using a finite amount of memory (e.g., see [17]).

An SDF graph, $G = (V, E)$, is a directed graph in which each vertex (*actor*) $v \in V$ represents a computational module for executing a given task, and each directed edge, $e \in E$, represents a first-in-first-out (FIFO) buffer for holding data values (*tokens*) and imposing data dependencies. SDF graphs represent computations that are executed iteratively on data streams that typically contain large, often unbounded numbers numbers of data samples. Each SDF actor therefore executes through a sequence of executions (invocations) as the enclosing SDF graph operates. An actor

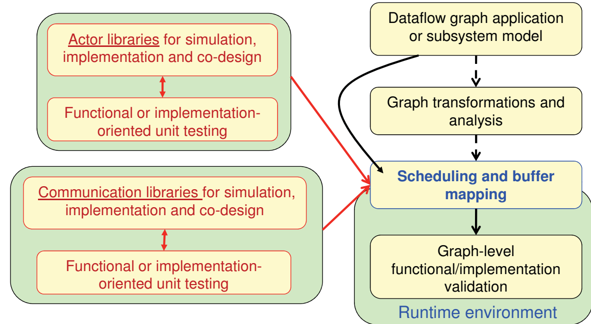

Figure 1: Proposed dataflow-based design approach.

is enabled to execute its computational task (i.e., its next invocation) only if sufficient numbers of tokens are available on its input edges. An actor produces certain numbers of tokens on its output edges as it executes. For proper operation, the output edges onto which an actor invocation produces data must have sufficient numbers of "empty spaces" (vacant positions in the corresponding FIFO buffers).

There is a wide variety of development environments that utilize dataflow models to aid in the design and implementation of DSP applications (e.g., see [7], [10], [13], [9]). In these tools, an application designer is able to develop and describe the functionality of an actor, and capabilities are provided for automated system simulation or synthesis. Some of these tools employ graphical user interfaces and support restricted models of computations, such as SDF.

3 Proposed Design Flow

Fig. 1 shows the overall design flow of the proposed lightweight dataflow approach for wireless communication and software-defined radio systems. Given an SDR application modeled by dataflow, in our approach, designers should be able to easily create actor and FIFO (dataflow graph edge) implementations along with associated test suites by following the proposed design methods associated with the LWDF programming model. Furthermore, by utilizing the LWDF runtime environment, designers can validate functional correctness when implementing an application on a specific target platforms.

Due to the design orthogonality among user-designed libraries and the runtime environment, design optimizations for applications can be explored at a high-level (i.e., inter-actor level) with full interoperability in relation to lower level optimizations provided by the back-end, platform-specific tools.

4 Lightweight Dataflow Programming Model

We propose a new dataflow programming model for design and implementation of DSP-based SDR systems. In this model, an each actor has an *operational context* (OC), which encapsulates parameters (P), local variables (θ_l) (variables whose values do not persist across firings), and state variables (θ_s) (variables whose values do persist across firings). The OC for an actor also contains references to the FIFOs corresponding to the input (R_i) and output (R_o) ports (edge connections) of the actor as a component of the enclosing dataflow graph, as well as a reference to the execution function (R_ξ) of the actor. That is, given an actor $v \in V$ in a dataflow graph, the operational context for v is defined as

$$OC(v) = \{R_i\} \cup \{R_o\} \cup \{\theta_l\} \cup \{\theta_s\} \cup R_\xi. \quad (1)$$

4.1 Design method of dataflow actors

Our proposed method for the design of dataflow actors in LWDF is described below. Based on different dataflow models of computation, the design can be adjusted accordingly in terms of the operational sequence associated with an actor firing.

Actor design in LWDF includes three interface functions — the *construct*, *execute*, and *terminate* functions. The *construct* function connects an actor to its input and output edges (FIFO channels), and performs any other pre-execution initialization associated with the actor. Similarly, the terminate function performs any operations that are required for "closing out" the actor after the enclosing graph has finished executing (e.g., deallocation of actor-specific memory or closing of associated files.

In the execute function, the operational sequence associated with an actor firing is implemented. Here, various restrictions are imposed on dataflow execution to make execution more predictable and programming more structured, while conforming to formal dataflow semantics. In particular, we impose the restriction that an actor firing proceeds through a sequence of *steps*. Such steps are loosely analogous to the "phases" of cyclo-static dataflow (CSDF) actors [5]. However, lightweight dataflow steps are significantly more flexible than CSDF phases in that they can reconfigure the behavior of subsequent steps (through changes to associated parameters or variables in the actor context) within a given firing. Such a reconfigurable step-based approach leads to a Turing complete programming approach in which arbitrary computations can be embedded, and a wide variety of specialized dataflow models can be accommodated as special cases.

When the overall scheduler (lightweight dataflow runtime environment) executes an actor firing, it starts at a common initial step, which we call the `FIRING_START` step, and keeps executing steps sequentially until one of the following happens.

- the current step gets suspended because required input data is not available,
- the current step gets suspended because required output buffer space is not available, or
- the last step in the enclosing firing (called the `FIRING_DONE` step), is reached.

Thus, `FIRING_START` and `FIRING_DONE` are two common steps for all actors, and monitoring of the other steps can be used as a way to gauge how far a given actor firing has progressed within its current firing (e.g., for real-time analysis or dynamic scheduling purposes).

4.2 Hardware/software retargetability

As an example of applying the LWDF programming model for DSP software design using C, a C-based LWDF actor can be implemented as an abstract data type (ADT) to enable efficient and convenient reuse of the actor across arbitrary applications. In typical C implementations, ADT components include header files to represent definitions that are exported to application developers and implementation files that contain implementation-specific definitions. We refer to this integration of LWDF and C as *LWDF-C*. In Section 5, we will show a design example for implementing a dataflow actor using LWDF-C.

To target an LWDF design for FPGA implementation using the Verilog hardware description language (HDL), we design a dataflow actor as a Verilog module, where input and output ports of the actor are mapped to unidirectional module ports. Such an actor-level module can contain arbitrary sub-modules for complex actors, and the functionality of an actor can in general be described in either a behavioral or structural style. Similarly, we refer to this integration of LWDF with Verilog as *LWDF-V*.

A finite state machine (FSM) is designed within an actor as a controller for implementing the associated *step transition graph* that represents the firing sequence of that actor. A step transition graph can be viewed as a restricted form of FSM where each LWDF actor step corresponds to a vertex (actor firing state), and a directed edge from one step s_1 to another step s_2

indicates that actor firing state can transition from s_1 to s_2 unconditionally or under some conditions in terms of the input data and values in the operational context.

For example, the one-hot strategy [6] is a natural state encoding in order to simplify the state decoding logic in the associated FSM. In general, a one-hot strategy is well suited to implementing step transition graphs where the number of steps is not excessive and area constraints are not extremely tight.

4.3 Orthogonal design for dataflow FIFOs

FIFO design for dataflow edge implementation is orthogonal to the design of dataflow actor in LWDF. That is, by using LWDF, application designers can focus on design of actors and mapping of edges to lower level communication protocols through separate design processes (if desired) and integrate them later through well-defined interfaces. Such design flow separation is useful due to the orthogonal objectives, which center around computation and communication, respectively, associated with actor and FIFO implementation.

FIFO design in LWDF typically involves different structures in software compared to hardware. For software design in C, tokens can have arbitrary types associated with them — e.g., tokens can be integers, floating point values, characters, or pointers (to any kind of data). Such an organization allows for flexibility in storing different kinds of data values, and efficiency in storing the data values directly (i.e., without being encapsulated in any sort of higher-level "token" object). For hardware design in HDLs, a dataflow graph edge is typically mapped to a FIFO module, and designers should have mechanisms for developing interfaces between actor and FIFO modules. For maximum flexibility in design optimization, LWDF provides for efficient retargetability of actor-FIFO interfaces across synchronous, asynchronous, and mixed-clock implementation styles.

4.4 Integration with the DICE unit testing framework

The *DSPCAD Integrative Command-line Environment* (*DICE*) is a package of utilities that facilities efficient management of software projects, especially for cross-platform design, and provides a lightweight, flexible, and language-agnostic unit testing environment [3]. Building on the modularity of dataflow based design, and the key LWDF feature of design orthogonality between dataflow actor and edge (FIFO), actors and FIFO implementations in LWDF can be seamlessly integrated with the DICE software package for

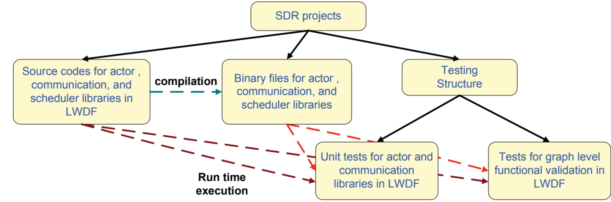

Figure 2: Design hierarchies for implementing dataflow graph components in LWDF using DICE.

unit testing. Such integration allows for efficiently configurable, cross-platform unit testing of LWDF actors and edges — e.g., across hardware and software actor implementations, or different types of communication mechanisms, such as block RAM, distributed or shared memories, for FIFO implementation.

Fig. 2 illustrates design hierarchies for implementing unit tests for dataflow graph components in LWDF using the DICE unit testing framework.

5 Design Example: A Reconfigurable Phase-Shift Keying System

In this section, we present a reconfigurable phase-shift keying (RPSK) system as an SDR application example using the proposed approach. PSK is a digital modulation technique in which patterns of data bits form data symbols, and are mapped to corresponding phase status configurations, which are then conveyed by modulating the phase of a carrier wave. In an agile SDR environment, RPSK is a useful subsystem because it allows the specific form of PSK employed to be dynamically adapted based on the existing status of the communication channel.

Fig. 3 shows a dataflow model of our targeted RPSK application. Three different modulation schemes can be configured in this application. They are binary PSK (BPSK), quadrature PSK (QPSK), and 8PSK.

Among these, BPSK is the most robust modulation scheme in the presence of high levels of noise in the communication channel. On the other hand, QPSK and 8PSK provide decreasing levels of robustness, but offer faster levels of data transfer and lower levels of receiver and transmitter energy requirements when channel conditions are better.

As shown in Fig. 3, actors T1 and T2 are table lookup actors used in the modulator and demodulator, respectively, for converting between binary sequences and phase status configurations. Each of these actor has a parameters M, which can be configured to select either BPSK ($M = 1$), QPSK ($M = 2$), or 8PSK ($M = 3$) as the modulation and demodulation schemes.

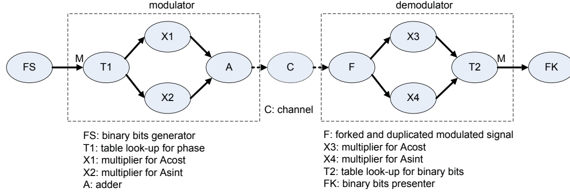

Figure 3: Dataflow graph for the RPSK application.

The parameter M also specifies the consumption rate of T1, and production rate of T2.

5.1 Simulation based on LWDF-C

Functional correctness of the LWDF design for the RPSK application can be verified by simulating its LWDF-C implementation using a simple scheduling strategy, which is an adaptation of the *canonical scheduling strategy* of the *functional DIF* environment [15]. Since the semantics of LWDF dataflow guarantee deterministic operation (i.e., the same input/output behavior regardless of the schedule that is applied), validation under such a simulation guarantees correct operation regardless of the specific scheduling strategy that is ultimately used in the final implementation. Such a simulation approach is therefore useful to orthogonalize functional validation of an LWDF design before exploring platform specific-schedule optimizations.

Fig. 4 shows LWDF-C implementation of the table-lookup actor in the RPSK application. Due to space limitations, we omit implementation details for the other actors in this application example.

In our experiments with our LWDF-C implementation, we simulated operation of the RPSK system on a bit stream consisting of 10,000 bits. This simulation was carried out in approximately 1.5 seconds on an 3GHz Intel Pentium PC with 2GB of RAM.

5.2 FPGA implementation based on LWDF-V

After validating functional correctness using LWDF-C, we manually implemented the RPSK application by applying LWDF-V together with a self-timed scheduling strategy. In such a self-timed implementation, an actor module fires whenever it has sufficient tokens (as required by the corresponding firing) available on its input FIFOs. The synthesis result derived from our LWDF implementation of the RPSK application uses 1,484 LUTs and 1,464 CLBs on a Xilinx Virtex-4 FPGA, which correspond to approximately 5% and 10% of the total available LUT and CLB resources, respectively, on the targeted FPGA device.

```
#include <stdio.h>
#include <stdlib.h>

#include "table_lookup.h"
#include "util.h"

table_lookup_context_type *table_lookup_new(file *file, int size,
        disps_fifo_pointer in, disps_fifo_pointer out) {
    table_lookup_context_type *context = null;
    float data = 0;
    int i = 0;

    if (size <= 0) {
        fprintf(stderr, "table_lookup_new error: invalid size");
        exit(1);
    }
    context = util_malloc(sizeof(table_lookup_context_type));
    context->table = util_malloc(size * sizeof(int));
    context->size = size;
    context->execute = (actor_execution_function_type)table_lookup_execute;

    for (i = 0; i < size; i++) {
        if (fscanf(file, "%f", &data) != 1) {
            fprintf(stderr, "table_lookup_new error: integer expected");
            exit(1);
        }
        (context->table)[i] = data;
    }

    context->status = actor_new_firing;
    context->in = in;
    context->out = out;

    return context;
}

void table_lookup_execute(table_lookup_context_type *context) {
    float result = 0;

    switch (context->status) {
    case actor_new_firing:
        if (disps_fifo_population(context->in) == 0) {
            return;
        }
        disps_fifo_read(context->in, &(context->index));

        if (((context->index) < 0) || ((context->index) >= (context->size))) {
            context->status = TABLE_LOOKUP_ERROR_INDEX;
            return;
        }

        context->status = TABLE_LOOKUP_INDEX_READ;
        /* falls through */
    case TABLE_LOOKUP_INDEX_READ:
        if (disps_fifo_population(context->out) >=
                disps_fifo_capacity(context->out)) {
            return;
        }
        result = (context->table)[context->index];
        disps_fifo_write(context->out, &result);
        context->status = ACTOR_FIRING_DONE;
        /* falls through */
    case ACTOR_FIRING_DONE:
        return;
    default:
        context->status = TABLE_LOOKUP_ERROR_ENTRY;
        break;
    }
}

void table_lookup_terminate(table_lookup_context_type *context) {
    free(context->table);
    free(context);
}
```

Figure 4: LWDF-C implementation of the table-lookup actor in the RPSK application.

Fig. 5 illustrates a design example of an actor template in our application of LWDF-V. By following such a structured format, designers can focus on implementing the core functionality of each actor for rapid prototyping, systematic validation, and efficient exploration of intra-actor optimization alternatives.

As shown in Fig. 5, actor_controller is a step transition controller, which implements a one-hot state machine and can be used, through appropriate parameterization, within all of the LWDF-V actor implementations. STEP_COUNT represents the number of bits used to store the state, and each state corresponds to a unique bit in the state storage. The *next* input gives the number of bits to "left_shift" the state. Thus, $next = N$ means that we want to "skip" $N-1$ states, and $next = 0$ means that we stay in the same state.

```
/* Common actor steps */
`define FS 4'b0001

/* User-defined number of steps */
`define STEP_COUNT 4

/* User-defined step symbols */
`define S2 4'b0010
`define S3 4'b0100
`define FD 4'b1000

/* Step transition controller implemented as a
   one hot state machine.
 */
module actor_controller(state, next, clock, reset);
   parameter STEP_COUNT = 1;
   parameter NEXT_LENGTH = 1;

   output [STEP_COUNT - 1 : 0] state;

   input [NEXT_LENGTH - 1 : 0] next;
   input clock;
   input reset;

   reg [STEP_COUNT - 1 : 0] state;

   always @(posedge clock or negedge reset)
   begin
      if (~reset) begin
         state <= 1;
      end
      else begin
         state <= state << next;
      end
   end
endmodule
```

```
/* actor design template */
module actor_name([port_list]);
   /* parameter declarations */
   parameter …
   /* port declarations */
   output …
   input …
   /* wire/reg declarations */
   reg …
   wire …

/* Verilog structural modeling:
   instantiation of actor's controller
 */
    actor_controller fsm(post_list);

/* Verilog behavior modeling */
/* sequential logics based on step
  transition
 */
always@(posedge clock) begin
   case (state)
      `FS:
      `S2:
      `S3:
      `FD:
   endcase
end

 /* combinational logics based on step
   transitions
  */
always@(*) begin
   case (state)
      `FS:
      `S2:
      `S3:
      `FD:
   endcase
endmodule
```

Figure 5: Design template for dataflow actor in LWDF-V.

6 Conclusion

In this paper, we have introduced a novel lightweight dataflow programming approach for design and implementation of software-defined radio systems. We provided a comprehensive specification of the proposed methodology, and demonstrated it concretely with actor implementation examples in C and Verilog that are oriented for fast simulation, embedded software realization, and FPGA mapping. We also demonstrated an SDR application example of a reconfigurable phase-shift keying (RPSK) system, which is a useful subsystem for agile SDR environments. Experimental results on our RPSK system are presented to demonstrate efficient high-level simulation based on C, and retargetability from C to Verilog based on the lightweight programming interface of LWDF, and a well-defined, structural HDL actor design methodology.

References

[1] H. Arslan. *Cognitive Radio, Software Defined Radio, and Adaptive Wireless Systems*. Springer, Septamber 2007.

[2] S. S. Bhattacharyya. Hardware/software co-synthesis of DSP systems. In Y. H. Hu, editor, *Programmable Digital Signal Processors: Architecture, Programming, and Applications*, pages 333–378. Marcel Dekker, Inc., 2002.

[3] S. S. Bhattacharyya, S. Kedilaya, W. Plishker, N. Sane, C. Shen, and G. Zaki. The DSPCAD integrative command line environment: Introduction to DICE version 1. Technical Report UMIACS-TR-2009-13, Institute for Advanced Computer Studies, University of Maryland at College Park, August 2009.

[4] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 21(2):151–166, June 1999.

[5] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, February 1996.

[6] M. D. Ciletti. *Advanced Digital Design with the Verilog HDL*. Prentice Hall, January 2010.

[7] J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[8] P. N. Garner and J. Dines. Tracter: A lightweight dataflow framework. Technical Report Idiap-RR-10-2010, Idiap Research Institute, May 2010.

[9] C. Hsu, M. Ko, and S. S. Bhattacharyya. Software synthesis from the dataflow interchange format. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, pages 37–49, Dallas, Texas, September 2005.

[10] G. Johnson. *LabVIEW Graphical Programming : Practical Applications in Instrumentation and Control*. McGraw-Hill, June 1997.

[11] E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[12] Y. Lin, R. Mullenix, M. Woh, S. Mahlke, T. Mudge, A. Reid, and K. Flautner. Spex: A programming language for software defined radio. In *Software Defined Radio Technical Conference and Product Exposition*, pages 13–17, 2006.

[13] J. L. Pino, K. Kalbasi, H. Packard, and E. Division. Cosimulating synchronous dsp applications with analog rf circuits. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, November 1998.

[14] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya. Functional DIF for rapid prototyping. In *Proceedings of the International Symposium on Rapid System Prototyping*, pages 17–23, Monterey, California, June 2008.

[15] W. Plishker, N. Sane, M. Kiemb, and S. S. Bhattacharyya. Heterogeneous design in functional DIF. In *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, pages 157–166, Samos, Greece, July 2008.

[16] W. Plishker, C. Shen, S. S. Bhattacharyya, G. Zaki, S. Kedilaya, N. Sane, K. Sudusinghe, T. Gregerson, J. Liu, and M. Schulte. Model-based DSP implementation on FPGAs. In *Proceedings of the International Symposium on Rapid System Prototyping*, Fairfax, Virginia, June 2010. Invited paper.

[17] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, second edition, 2009.