A Lightweight Dataflow Approach for Design and Implementation of SDR Systems

Chung-Ching Shen, *William Plishker, Hsiang-Huang Wu, and Shuvra S. Bhattacharyya*

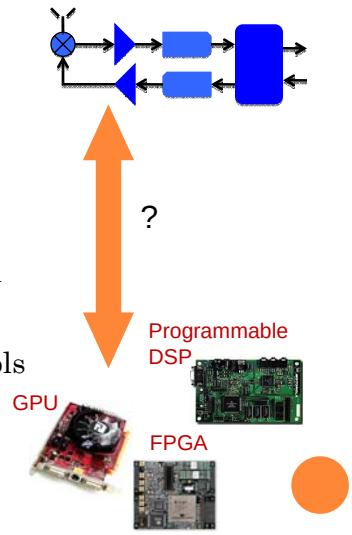
Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies,
University of Maryland, College Park, MD

OUTLINE

- Introduction
- Background
- Dataflow-based Design Flow
- Lightweight Dataflow Programming Approach
- Design Example
- Conclusion and Future Work

INTRODUCTION

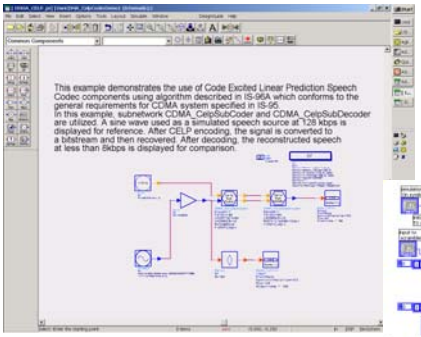
- In SDR, designers attempt to
  - Implement most of the complex signal handling of radio using software
  - Rapidly target a variety of platforms (retargetability)
- In modern, complex systems we would like to
  - Create an application description independent of the targeted platform
  - Interface with a diverse set of tools
  - Arrive at an initial prototype quickly
  - Achieve high performance, less resource usage



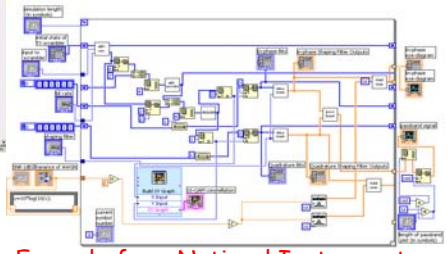
MODEL-BASED DESIGN BASED ON DATAFLOW MODEL OF COMPUTATION

- Dataflow-based design is a specific form of *model-based design*
  - i.e., high level application subsystems are specified in terms of components that interact through formal models of computation
- Dataflow model of computation
  - Used widely in design tools for DSP
  - Used widely for expressing the functionality of DSP applications
  - Application is modeled as a directed graph
  - Data-driven execution model
  - Iterative execution

EXAMPLE: DATAFLOW-BASED DESIGN TOOLS FOR DSP



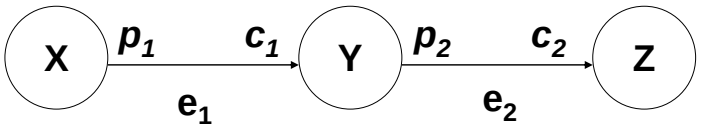
Example from Agilent ADS tool



Example from National Instruments LabVIEW

DATAFLOW GRAPHS AND SEMANTICS

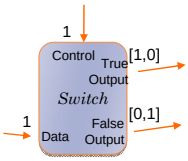
- Vertices (actors) represent computation
- Edges represent FIFO buffers
- Edges may have delays, implemented as initial tokens
- Tokens are produced and consumed on edges
- Different models have different rules for production and consumption (SDF=fixed, CSDF=periodic, BDF=dynamic)



EXAMPLE: CORE FUNCTIONAL DATAFLOW (CFDF) MODEL OF COMPUTATION

- Divide actors into a set of *modes*
  - Each mode has a fixed consumption and production behavior, but actors may dynamically switch between modes.
- Write the enabling conditions for each mode
- Write the computation associated with each mode
  - Including next mode to *enable* and then *invoke*
- For example, consider a standard Switch:

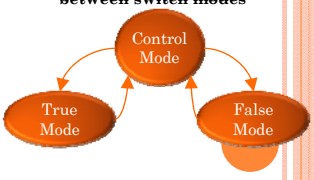
Switch Actor



Production & consumption behavior of switch modes

Mode	Consumes		Produces	
	Control	Data	True	False
Control	1	0	0	0
True	0	1	1	0
False	0	1	0	1

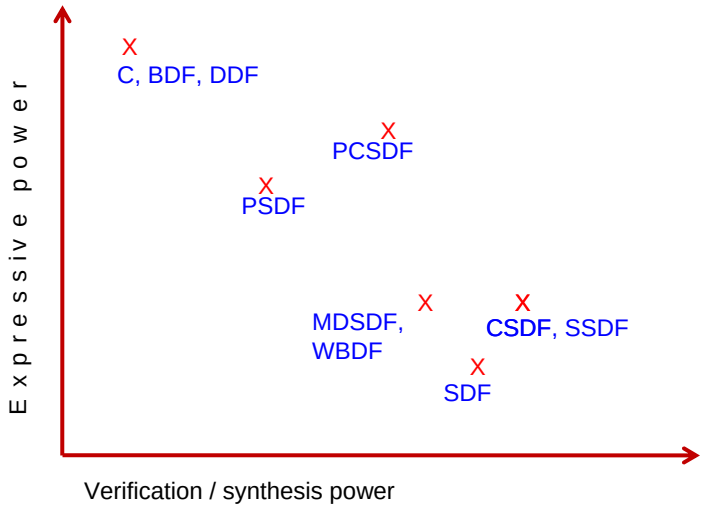
Mode transition diagram between switch modes



EVOLUTION OF DATAFLOW MODELS OF COMPUTATION FOR DSP: EXAMPLES

- **Computation Graphs and Marked Graphs [Karp 1966, Reiter 1968]**
- **Synchronous dataflow, [Lee 1987]**
  - **Static multirate behavior**
  - **SPW (Cadence) , National Instruments LabVIEW, and others.**
- **Well behaved stream flow graphs [1992]**
  - **Schemas for bounded dynamics**
- **Boolean/integer dataflow [Buck 1994]**
  - **Turing complete models**
- **Multidimensional synchronous dataflow [Lee 1992]**
  - **Image and video processing**
- **Scalable synchronous dataflow [Ritz 1993]**
  - **Block processing**
  - **COSSAP (Synopsys)**
- **CAL [Eker 2003]**
  - **Actor-based dataflow language**
- **Cyclo-static dataflow [Bilsen 1996]**
  - **Phased behavior**
  - **Eonic Virtuoso Synchro, Synopsys El Greco and Cocentric, Angeles System Canvas**
- **Bounded dynamic dataflow**
  - **Bounded dynamic data transfer [Pankert 1994]**
- **The processing graph method [Stevens, 1997]**
  - **Reconfigurable dynamic dataflow**
  - **U. S. Naval Research Lab, MCCI Autocoding Toolset**
- **Stream-based functions [Kienhuis 2001]**
- **Parameterized dataflow [Bhattacharya 2001]**
  - **Reconfigurable static dataflow**
  - **Meta-modeling for more general dataflow graph reconfiguration**
- **Reactive process networks [Geilen 2004]**
- **Blocked dataflow [Ko 2005]**
  - **Image and video through parameterized processing**
- **Windowed synchronous dataflow [Keinert 2006]**
- **Parameterized stream-based functions [Nikolov 2008]**
- **Enable-invoke dataflow [Plishker 2008]**
- **Variable rate dataflow [Wiggers 2008]**

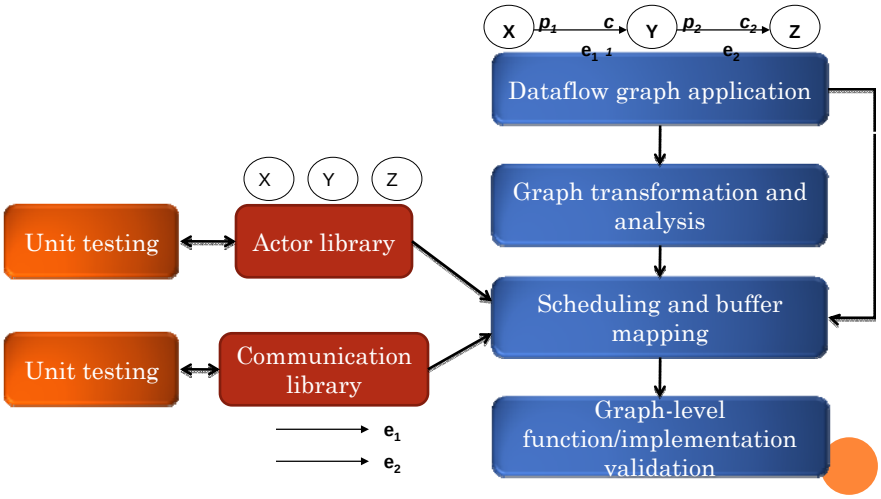
MODELING DESIGN SPACE OF DATAFLOW



OUTLINE

- Introduction
- Background
- Dataflow-based Design Flow
- Lightweight Dataflow Programming Approach
- Design Example
- Conclusion and Future Work

DESIGN FLOW



OUTLINE

- Introduction
- Background
- Dataflow-based Design Flow
- Lightweight Dataflow Programming Approach
- Design Example
- Conclusion and Future Work

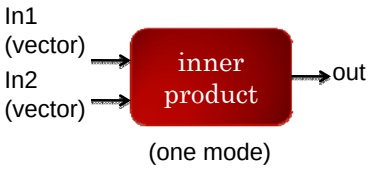
LIGHTWEIGHT DATAFLOW PROGRAMMING APPROACH

- LWDF
- A dataflow programming approach for model-based design and implementation of DSP systems.
  - By ``lightweight'': minimally intrusive on existing design processes, and require minimal dependence on specialized tools or libraries.
- Features
  - Improve the **productivity** of the design process and the quality of derived implementations.
  - **Retargetability** across different platforms.
  - Allow designers to **integrate and experiment with dataflow modeling approaches** relatively quickly and flexibly into existing design methodologies and processes.

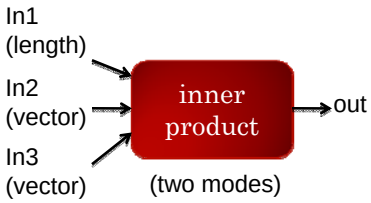
LWDF DESIGN PRINCIPLES I

- Each actor has an operational context (OC), which encapsulates
  - parameters
  - mode/status variables
  - local variables
  - references to the FIFOs
    - corresponding to the input and output ports of the actor as a component of the enclosing dataflow graph.
  - reference to the execution functions of the actor.

OPERATIONAL CONTEXT – EXAMPLE OF LWDF-C



(one mode)



(two modes)

```
typedef struct {

    /* parameters */
    int length;

    /* mode variable */
    int mode;

    /* refs to the FIFOs */
    fifo_pointer in1;
    fifo_pointer in2;
    fifo_pointer out;

    /* refs to the execution functions*/
    actor_enable_function_type enable;
    actor_invoke_function_type invoke;

} inner_product_context_type;
```

```
typedef struct {

    /* local variables */
    int length;

    /* mode variable */
    int mode;

    /* refs to the FIFOs */
    fifo_pointer in1;
    fifo_pointer in2;
    fifo_pointer in3;
    fifo_pointer out;

    /* refs to the execution functions*/
    actor_enable_function_type enable;
    actor_invoke_function_type invoke;

} inner_product_context_type;
```

LWDF DESIGN PRINCIPLES II

- Methods that are involved for the implementation of an actor
  - ***Construct***: connects an actor to its input and output edges (FIFO channels), and performs any other pre-execution initialization associated with the actor.
  - ***Execute***: implements the operational semantics of a dataflow model associated with an actor firing.
    - e.g, enable and invoke functions of CFDF
  - ***Terminate***. performs any operations that are required for ``closing out'' the actor after the enclosing graph has finished executing

METHODS – EXAMPLE OF LWDF-C


```
void inner_product_invoke(inner_product_context_type *context) {
    int i = 0;
    int sum = 0;
    int x_value = 0;
    int y_value = 0;
    switch (context->mode) {
        case STORE_LENGTH:
            fifo_read(context->in1, &context->length);
            if (context->length < 0) {
                context->mode = INNER_PRODUCT_ERROR_LENGTH;
                return;
            }
            context->mode = PROCESS;
            break;
        case PROCESS:
            sum = 0;
            for (i = 0; i < context->length; i++) {
                fifo_read(context->in1, &(x_value));
                fifo_read(context->in2, &(y_value));
                sum += (x_value * y_value);
            }
            fifo_write(context->out, &sum);
            context->mode = STORE_LENGTH;
            break;
        default:
            context->mode = INNER_PRODUCT_ERROR_INVALID;
            break;
    }
    return;
}
```

ACTOR APIS IN LWDF-C

Type Definitions:

```
/* An actor's operational context (OC). */
typedef struct {
    /* parameters */
    /* local and mode variables */
    /* references to FIFO pointers */
    /* reference to a pointer of actor's execution functions */
} [actor_name]_context_struct;

/* A pointer to actor enable/invoke functions, which are functions that
    executes an actor with a given context. */
typedef void (*actor_[enable/invoke]_function_type) (struct
    actor_context_struct *context);
```

Key Methods:

```
[actor_name]_context_type *[actor_name]_new(…);
void [actor_name]  enable([actor_name]_context_type *context);
void [actor_name]  invoke([actor_name]_context_type *context);
void [actor_name]  terminate([actor_name]_context_type *context);
```

ORTHOGONALIZING DATAFLOW COMMUNICATION: FIFO

- Each data item in the FIFO is referred to as a "token", and tokens can have arbitrary types associated with them.
- For a given FIFO instance, there is a fixed token size (number of bytes per token).
- Tokens have arbitrary types
  - e.g., integers, floating point values (float or double), characters, or pointers (to any kind of data).

FIFO APIS IN LWDF-C

Type Definitions:

```
/* A FIFO. */
typedef struct _fifo_struct fifo_type;
/* A pointer to a fifo. */
typedef fifo_type *fifo_pointer;
```

Key Methods:

```
fifo_pointer fifo_new(int capacity, int token_size);
int fifo_population(fifo_pointer fifo);
int fifo_capacity(fifo_pointer fifo);
void fifo_write(fifo_pointer fifo, void *data);
void fifo_write_block(fifo_pointer fifo, void *data, int size);
void fifo_read(fifo_pointer fifo, void *data);
void fifo_read_block(fifo_pointer fifo, void *data, int size);
```

IMPLEMENTATION IN VERILOG

- LWDF-V
- Dataflow actor ->Verilog module
  - This module can contain arbitrary sub-modules for complex actors
  - This module can be behavioral or structural
  - Actor port -> module port
  - Actor constructor -> module instantiation
- Step/Mode transition graph -> FSM within actor
  - Controls actor execute function
  - One or more concurrent processes that make up the steady-state behavior of the actor

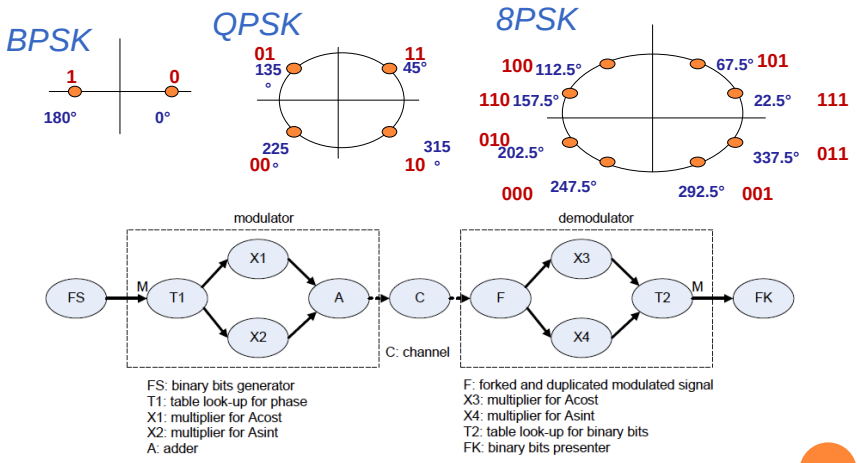
EXAMPLE: IMPLEMENTATION TEMPLATE IN LWDF-V

```
`include "actor_modules.v"
`include "libs.v"

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

/* Number of steps */
`define STEP_COUNT 4

`define S2 4'b0010
`define S3 4'b0100
`define FD 4'b1000

module actor_name(port_list);
   //parameters
   parameter …
   //output and input ports
   output …
   input …

   //internal registers and wires
   //Verilog structural modeling
    //Instantiation of the step transition controller
    actor_controller fsm(post_list);

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

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

OUTLINE

- Introduction
- Background
- Dataflow-based Design Flow
- Lightweight Dataflow Programming Approach
- Design Example
- Conclusion and Future Work

DESIGN EXAMPLE: RECONFIGURABLE PHASE-SHIFT KEYING



- M=1 for BPSK, M=2 for QPSK, and M=3 for 8PSK

DESIGN EXAMPLE: RECONFIGURABLE PHASE-SHIFT KEYING

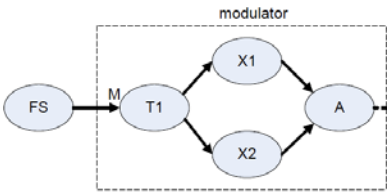
LWDF-C implementation for the table-lookup actor in RPSK

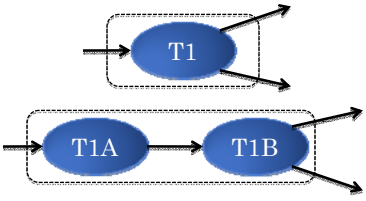
```
#include <stdio.h>
#include <stdlib.h>

#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;
   }
}
```

DESIGN EXAMPLE: RECONFIGURABLE PHASE-SHIFT KEYING

Graph topology designed in LWDF-C (i.e., example of RPSK modulator)





actors[ACTOR_BINARY_SIGNAL_SOURCE] = (actor_context_type *)(file_source_new(in_file, fifo2));
actors[ACTOR_PSK_PACK] = (actor_context_type *)(psk_pack_new(fifo1, fifo2, fifo3));
actors[ACTOR_TABLE_LOOKUP] = (actor_context_type*)(table_lookup_new(table_file, table_size, fifo3, fifo4));
actors[ACTOR_FORK] = (actor_context_type *)(fork_new(fifo4, fifo5, fifo6));
actors[ACTOR_CARRIER_FREQ_MUL1] = (actor_context_type*)(carrier_freq_mul_new(CARRIER_FREQ_MUL_TYPE_FORWARD, CARRIER_FREQ_TYPE_COS, INIT_AMPLITUDE, CARRIER_FREQ, COS_SIGN, DEMODULATE_PTS, fifo5, fifo7));
actors[ACTOR_CARRIER_FREQ_MUL2] = (actor_context_type* (carrier_freq_mul_new(CARRIER_FREQ_MUL_TYPE_FORWARD, CARRIER_FREQ_TYPE_SIN, INIT_AMPLITUDE, CARRIER_FREQ, SIN_SIGN, DEMODULATE_PTS, fifo6, fifo8));
actors[ACTOR_PSK_ADD] = (actor_context_type*)(psk_add_new(500e-6, fifo7, fifo8, fifo9));
actors[ACTOR_FILE_SINK] = (actor_context_type *)(file_sink_float_new(out_file, fifo9));

DESIGN EXAMPLE: RECONFIGURABLE PHASE-SHIFT KEYING

Application simulation driven by a simple scheduler

```
void util_simple_scheduler(actor_context_type *actors[], int actor_count,
        char *descriptors[]) {
        boolean progress = FALSE;
         int i = 0;
        do {
                progress = 0;
                for (i = 0; i < actor_count; i++) {
                        progress |= util_guarded_execution(actors[i], descriptors[i]);
                }
        } while (progress);
}
```

guarded execution for CFDF model

- Simulation based on LWDF-C
  - 3GHz Intel Pentium PC with 2GB of RAM
  - Input bit stream: 10,000 bits
  - Simulation time: 1.5 seconds

DESIGN EXAMPLE: RECONFIGURABLE PHASE-SHIFT KEYING

Graph topology designed in LWDF-V (i.e., example of RPSK modulator)

file_source #(.DATA_WIDTH(DATA_WIDTH), .IN_FILE(IN_FILE)) fs_module1(.out(fifo1_in), .write(fifo1_write), .out_full(fifo1_full), .clock(clock), .reset(reset), .enable_firing_reset(1'b1));
psk_table_lookup #(.BLOCK_SIZE(BLOCK_SIZE), .INPUT_DATA_WIDTH(DATA_WIDTH), .OUTPUT_DATA_WIDTH(DATA_WIDTH), .COUNTER_WIDTH(COUNTER_WIDTH)) psk_table_lookup_module (.out(fifo2_in), .read(fifo2_read), .write(fifo2_write), .in(fifo1_out), .in_empty(fifo1_empty), .out_full(fifo2_full), .clock(clock), .reset(reset), .enable_firing_reset(1'b1));

actor_fork #(.DATA_WIDTH(DATA_WIDTH)) fork_module (.out1(fifo3_in), .out2(fifo4_in), .read(fifo2_read), .write(fifo3_4_write), .in(fifo2_out), .in_empty(fifo2_empty), .out1_full(fifo3_full), .out2_full(fifo4_full), .clock(clock), .reset(reset), .enable_firing_reset(1'b1));

multiplier #(.INPUT_DATA_WIDTH(DATA_WIDTH), .OUTPUT_DATA_WIDTH(DATA_WIDTH)) mul_module1 (.out(fifo5_in), .read(fifo3_read), .write(fifo5_write), .in1(fifo3_out), .in2(1), .in1_empty(fifo3_empty), .in2_empty(0), .out_full(fifo5_full), .clock(clock), .reset(reset), .enable_firing_reset(1'b1));

multiplier #(.INPUT_DATA_WIDTH(DATA_WIDTH), .OUTPUT_DATA_WIDTH(DATA_WIDTH)) mul_module2 (.out(fifo6_in), .read(fifo4_read), .write(fifo6_write), .in1(fifo4_out), .in2(1), .in1_empty(fifo4_empty), .in2_empty(0), .out_full(fifo6_full), .clock(clock), .reset(reset), .enable_firing_reset(1'b1));

add #(.DATA_WIDTH(DATA_WIDTH)) add_module (.out(fifo7_in), .read(fifo5_6_read), .write(fifo7_write), .in1(fifo5_out), .in2(fifo6_out), .out_full(fifo7_full), .clock(clock), .reset(reset), .enable_firing_reset(1));

file_sink #(.DATA_WIDTH(DATA_WIDTH), .OUT_FILE(OUT_FILE)) fk_module(.read(fifo7_read), .in(fifo7_out), .in_empty(fifo7_empty), .clock(clock), .reset(reset), .enable_firing_reset(1'b1));

fifo #(.DATA_WIDTH(DATA_WIDTH), .CAPACITY(CAPACITY), .PTR_WIDTH(PTR_WIDTH)) fifo_module1 (fifo1_out, fifo1_full, fifo1_empty, fifo1_in, fifo1_read, fifo1_write, clock, clock, reset);

DESIGN EXAMPLE: RECONFIGURABLE PHASE-SHIFT KEYING

- Application execution driven by self-timed scheduling strategy
  - That is, an actor module fires whenever it has sufficient tokens available on its input FIFOs
- FPGA implementation based on LWDF-V
  - Target FPGA device: Xilinx Virtex-4
  - Resource utilizations after synthesis: 1,484 LUTs (5% util. rate) and 1,464 CLBs (10% util. rate)

CONCLUSION AND FUTURE WORK

- A lightweight dataflow programming approach for SDR systems
  - Simplified flow for design and testing
  - Well-structured design templates
- Retargetability: design and implementation in C and Verilog
  - Fast simulation, embedded software realization, and FPGA mapping
- Apply and experiment with various dataflow MoCs for more SDR applications

Thank You!