Implementing Latency-Insensitive Dataflow Blocks

Bingyi Cao
Computer Science
Columbia University
bingyi@cs.columbia.edu

Kenneth A. Ross
Computer Science
Columbia University
kar@cs.columbia.edu

Martha A. Kim
Computer Science
Columbia University
martha@cs.columbia.edu

Stephen A. Edwards
Computer Science
Columbia University
sedwards@cs.columbia.edu

***Abstract*—To simplify the implementation of dataflow systems in hardware, we present a technique for designing latency-insensitive dataflow blocks. We provide buffering with back-pressure, resulting in blocks that compose into deep, high-speed pipelines without introducing long combinational paths. Our input and output buffers are easy to assemble into simple unit-rate dataflow blocks, arbiters, and blocks for Kahn networks. We prove the correctness of our buffers, illustrate how they can be used to assemble arbitrary dataflow blocks, discuss pitfalls, and present experimental results that suggest our pipelines can operate at a high clock rate independent of length.**

1. Introduction

While integration levels and clock frequencies continue to soar, the speed of light has remained stubbornly constant, meaning circuits must limit themselves to "local" communication within a single clock cycle. Techniques such as wire pipelining and latency-insensitive design [4] provide ways to break long communication paths, but these techniques either require the designer to consider multicycle wire delays as part of the design process or introduce needless delays because the algorithm was developed assuming faster communication.

In this paper, we present a methodology based on patient input and output buffers that allows a designer to assemble large dataflow networks without introducing long wires that would reduce the clock frequency. In our approach, dataflow blocks can be designed naïvely: with arbitrary combinational logic and without buffering. Adding our buffers breaks any combinational paths from inputs to outputs, allowing blocks to be assembled into arbitrary networks without introducing any clock-rate-sapping long wires. Our technique enables high clock rates for the same reason as latency insensitive design, but because our technique exposes the dataflow model of computation, we can express more efficient algorithms that express richer communication patterns.

One of our basic goals is to reliably transfer a sequence of data tokens (such as a series of bytes) across a synchronous point-to-point link. Such a link needs flow control because we assume the transmitter may not have the next token ready or the receiver is not yet willing to consume a token. In these cycles, the receiver must wait and the transmitter must hold the token and transmit it later. We write "⊥" to denote the absence of a token. Our buffers ensure tokens are sent and received reliably between blocks, holding any tokens that are not ready to be consumed and exerting backpressure to ensure no data is lost or duplicated.

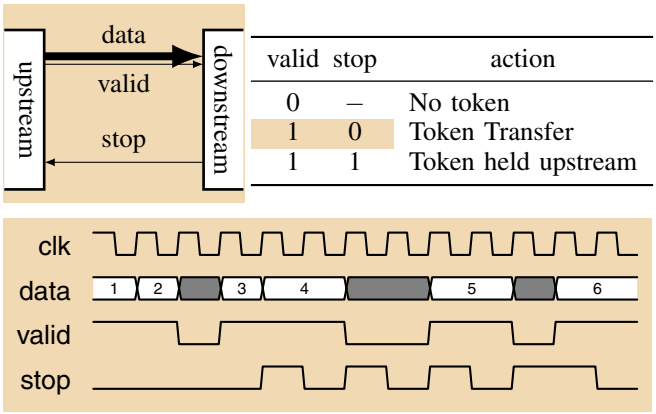
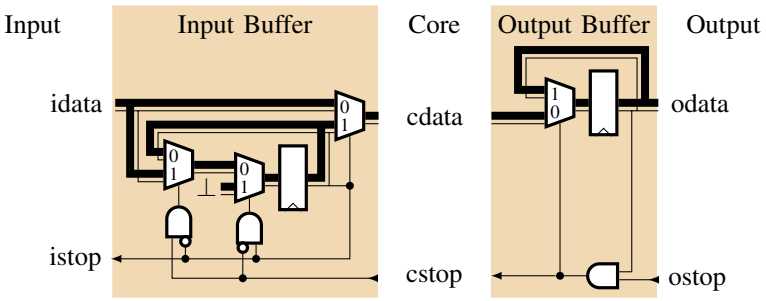


Figure 1. Our flow-control protocol transmitting the sequence of tokens 1,2,3,4,5,6. After Cortadella et al. [5] and Li et al. [13].

We begin by describing the well-known flow-control protocol we use: a channel consists of data, a *valid* bit that indicates data is present and must not be dropped, and a *stop* signal indicating backpressure. Then, we describe a pair of buffers that speak this protocol on both their inputs and outputs and prove that they preserve input sequences. We show how these buffers can be placed around a core—the computational logic of the dataflow block—to produce unit-rate and more complicated dataflow blocks. Finally, we illustrate our methodology with an application—range partitioning for databases—and show the maximum clock rate of the pipeline implemented on an FPGA is barely affected by pipeline length.

2. Flow Control

For flow control, we use the well-known synchronous parallel protocol in which a *valid* bit indicates when an upstream transmitter is attempting to send data and a *stop* bit indicates when a downstream receiver is unable to receive it (Figure 1). This protocol has been given various names:



Input		Core		Register		Action
data	stop	data	stop	current	next	
⊥	0	⊥	–	⊥	⊥	No-op
t	0	*t*	0	⊥	⊥	Pass
t	0	*t*	1	⊥	*t*	Buffer
–	1	*t*	1	*t*	*t*	Hold
–	1	*t*	0	*t*	⊥	Send

Core		Output			Action
data	stop	data	stop	next	
⊥	0	⊥	–	⊥	No-op
t	0	⊥	–	*t*	Buffer
–	1	*t*	1	*t*	Hold
⊥	0	*t*	0	⊥	Send
t	0	*t*′	0	*t*	Pipeline

```
module inbuf #(parameter W=8)
  (input        clk,
   input [W:0]  idata, output istop,
   output [W:0] cdata, input  cstop);

reg [W:0] ireg = {1'b0, {W{1'bx}}}; // Empty

assign istop = ireg[W];             // Stop if buffering
assign cdata = istop ? ireg : idata; // Send buffered

always @(posedge clk)
  if      (!cstop && ireg[W]) // Will core consume?
    ireg <= {1'b0, {W{1'bx}}}; // Yes: empty buffer
  else if ( cstop && !ireg[W]) // Core stop, empty?
    ireg <= idata;             // Yes: load buffer

endmodule
```

```
module outbuf #(parameter W=8)
  (input        clk,
   input [W:0]  cdata,
   output       cstop,
   output reg [W:0] odata,
   input        ostop);

 initial odata = {1'b0, {W{1'bx}}}; // Empty

// Stop the core when buffer full and output not ready
assign cstop = odata[W] && ostop;

always @(posedge clk)
  if (!cstop)          // Can we accept more data?
    odata <= cdata; // Yes: load the buffer

endmodule
```

Figure 2. Circuits, transition tables, and Verilog for our input and output buffers. *t* denotes a token: data whose valid bit is true; ⊥ denotes the absence of a token: data with a false valid bit, and – means the signal is ignored. Each highlighted entry in the transition tables indicates when a token is transferred. In the Verilog, *idata*, *cdata*, *buffer*, and *odata* consist of *W* data bits and one valid bit—the MSB.

Cortadella et al. [5] call it "SELF," and Li et al. [13] call it "LID-1SS." When the downstream receiver asserts the *stop* signal, it indicates the receiver is unable to capture the token in the current clock cycle, so the upstream transmitter needs to hold the token and attempt to retransmit it in the next cycle. In cycles where *valid* is true and *stop* is false, the receiver consumes the token.

We use this protocol because of its simplicity. It provides reliable in-order delivery of a single data stream even if the transmitter or receiver is unwilling to communicate because it is busy or its output channels are blocked.

3. Our Input and Output Buffers

One of our main contributions is a pair of input and output buffers that are easy to use to implement a variety of dataflow blocks that can be assembled into high-speed pipelines and richer networks. Both consist of a pair of input and output channels, each complying with the flow control protocol. The upstream backpressure output from the input buffer and the downstream data output from the output buffer are each registered (i.e., driven directly from flip-flops) so they can be chained together without introducing arbitrary long combinational paths. Furthermore, systems built from dataflow blocks that use our buffers do not need any global controller, so clock frequency of these systems is limited only by the slowest single block, not by the number of connected blocks.

Figure 2 shows our buffers. The input buffer is on the left; the output buffer is on the right, and "Core" denotes the location of the logic for the dataflow block. We discuss how to design the logic for a core starting in Section 4.

Each buffer has two ports—one input and one output—and each port uses the same flow-control protocol. The main difference between the two buffers is which paths are combinational: the input buffer has no combinational upstream path; the output buffer has no combinational downstream path. When used together, there is no combinational path in either direction, so connecting blocks does not affect the critical timing path.

In our buffers, we always treat data and its valid bit as a bundle, which makes our circuits easier to understand and verify. We draw these bundles as a thick line with a nearby thin line for the valid bit. By design, our buffers keep data linked to its valid bit to avoid inadvertently duplicating data.

The output buffer is the simpler of the two. It sends downstream data along with its valid bit from the output register. When the output register holds a token and the downstream backpressure signal *ostop* is asserted, the register holds its value. Otherwise, *cstop* is false and the output register is loaded. By design, there is no combinational path from the core to downstream to enable chaining.

The input buffer is more complicated. The *idata* input includes both data and a valid bit; *istop* is the upstream backpressure signal, driven directly by the valid bit in the input register. Complementing the output buffer, the input buffer has no combinational path from the core to the upstream stop signal (*cstop*), again to enable arbitrary chaining without negatively impacting clock frequency.

The input buffer has two states: empty or holding a token. When the register does not contain a token—its valid bit is false—no backpressure is applied: *istop* remains low, and a token on *idata* is routed directly to the core (*cdata*). If the core indicates it can consume the token by setting *cstop* false, the register holds its invalid contents. If instead the core asserts *cstop*, the register loads any incoming token on *idata* along with its valid bit.

If the register contains a token, *istop* is asserted and the token is routed to the core on *cdata*. If the core sets *cstop* false, the buffer will be emptied (denoted by the $\perp$ input to the mux) at the end of the cycle. Otherwise, the register reloads its value and retains the token into the next cycle.

3.1. Proofs of the Buffers' Correctness

Both input and output buffers preserve the sequence of tokens presented on their inputs, which we prove below. More specifically, no token sent or received by the protocol is dropped, duplicated, or delivered out-of-order.

Definitions: A *token* is data whose associated valid bit is true. The *input sequence* is the sequence of tokens on *idata* in cycles where *istop* is false (i.e., the sequence of consumed tokens); the *core sequence* is the sequence of tokens on *cdata* in cycles where *cstop* is false; the *output sequence* is the sequence of tokens on *odata* in cycles where *ostop* is false (i.e., the sequence of emitted tokens).

Theorem 1 If the core never blocks forever—after any cycle, *cstop* is eventually false—the input buffer ensures the core sequence is equal to the input sequence.

Proof We show that each token that appears on the input sequence appears exactly once in the core sequence and that tokens are not reordered. We take the behavior described in the input buffer transition table (left in Figure 2) as the definition of the input buffer. Note that for a token t to appear in a sequence, the *valid* bit must be true and *stop* bit must be false. These cases are highlighted in the table.

A token that appears in the input sequence appears exactly once in the core sequence because once it appears in the input sequence, it will eventually appear in the core sequence and then will never appear again because the input buffer no longer holds the information. More specifically, when a token appears in the input sequence, there are two cases: in the Pass case, the token that appeared in the input sequence appears immediately in the core sequence and is thereafter forgotten (the buffer will hold $\perp$ in the next cycle); in the Buffer case, no token appears in the core sequence in that cycle, but the token from the input sequence is stored in the buffer.

In cycles where a token is in the buffer, there are two cases: in the Hold case, no token appears on the core sequence and the token that is in the buffer is held there; in the Send case, the token in the buffer appears in the core sequence and is subsequently forgotten (the buffer will hold $\perp$ in the next cycle). By assumption, the Hold case cannot occur forever (since *cstop* eventually becomes false), so a token in the buffer always eventually appears in the core sequence.

Thus, once a token has appeared in the input sequence, it either appears immediately in the core sequence once, or it eventually appears in the core sequence once.

Furthermore, an input token cannot overtake another. When an input token appears, if it does not immediately appear in the core sequence, no token may appear in the input sequence until the cycle after the buffered token does appear in the core sequence. When a token is in the buffer, *istop* is asserted and by definition, no additional input token appears in the input sequence. Figure 3 illustrates this behavior. □

Theorem 2 If the downstream consumer never blocks forever—after any cycle, *ostop* is eventually false—the output buffer ensures the output sequence is equal to the core sequence.

Proof Like above, we show that each token that appears on the core sequence appears exactly once in the output sequence and that tokens are not reordered. Again, we reason with the behavior described in the output buffer transition table (right in Figure 2).

There are two possibilities when a token t appears in the core sequence. In the Buffer case, no token appears in the output sequence but the input token t will appear on the output data in the next cycle. In the Pipeline case, a distinct, previously buffered token appears in the output sequence in that cycle, but t will appear on the output data in the next cycle.

Three things may happen when a token t appears on the output data lines. In the Hold case, no token appears in the output sequence (because *stop* is asserted) and the token

Input Buffer

Input		Core	
data	stop	data	stop
t	0	t	0
t	0	t	1
–	0	t	0
t	0	t	1
–	1	t	1
⋮	⋮	⋮	⋮
–	1	t	1
–	1	t	0

Output Buffer

Core		Output	
data	stop	data	stop
t	0	$t'/\perp$	$0/-$
$t''/\perp$	$0/-$	t	0
t	0	$t'/\perp$	$0/-$
–	1	t	1
⋮	⋮	⋮	⋮
–	1	t	1
$t''/\perp$	$0/-$	t	0

Figure 3. Behavior of the input and output buffers: a visualization of the proofs of of Theorems 1 & 2. Time runs from top to bottom; columns indicate values on ports; highlighting indicates a token being transferred. The input buffer delivers a token immediately, in the next cycle, or once the core releases *cstop*. The output buffer delivers the token in the next cycle or once the downstream receiver release *ostop*. Tokens may also be transmitted in the first cycle of such a process or be received in the last cycle of such a process.

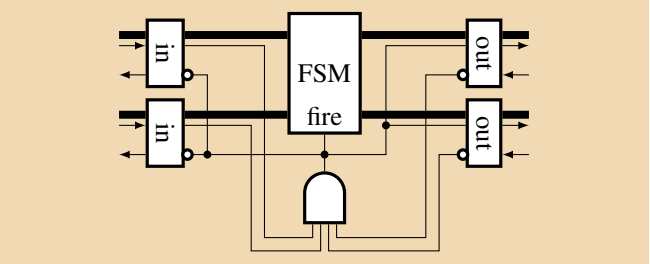


Figure 4. A two-input, two-output unit-rate dataflow block with state. The Mealy-style state machine must hold its state when *fire* is false. Note that the "stop" signals inside the core are active-low.

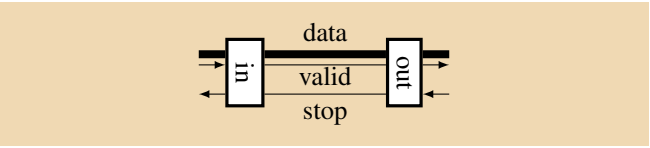


Figure 5. A single-place buffer built from a pair of our input and output buffers. This behaves like the relay station of Li et al. [13].

remains on the output data lines in the next cycle. In the Send case, the held token appears in the output sequence but is forgotten thereafter (the output data lines will be $\perp$ in the next cycle). In the Pipeline case, the held token appears in the output sequence, the output data lines will hold the (distinct) token that appeared in the input sequence in this cycle, and the held token will again be forgotten. By assumption, *ostop* is eventually false, so either Send or Pipeline will occur eventually.

Thus, once a token appears in the core sequence, that token will eventually appear in the output sequence. Also, the output buffer forgets the token after it appears in the output sequence.

Finally, tokens in the core sequence cannot overtake each other. When a token appears in the core sequence, the core sequence cannot have another token until the cycle in which the first token appears in the output sequence (the Pipeline case). Figure 3 illustrates this behavior. □

During development, we also ran our buffers on randomly generated test vectors (random data, valid, and stop bits), extracted the input and output sequences, and verified they were identical.

4. Core Logic: Unit-Rate Dataflow Blocks

Our buffers make it easy to build unit-rate dataflow blocks—blocks that require a token on each input before producing tokens on each output. Each such block consists of our input and output buffers around a core: the logic responsible for transforming inputs to outputs, which may be combinational or hold state, and an AND gate that detects when the block can consume inputs and produce outputs.

Figure 4 shows how to use our buffers to implement a two-input, two-output unit-rate dataflow block; more inputs and outputs follow the same pattern. The AND gate instructs the core to "fire"—consume inputs and produce outputs—when all the input tokens are present and none of the output buffers are blocked. The Mealy FSM should only advance its state when *fire* is true, perhaps by connecting *fire* to a latch enable input on every flip-flop in the core. A combinational block—a degenerate FSM—may ignore *fire*. In any case, the FSM need not suppress spurious outputs where *fire* is false because the output buffers ignore data (treat as invalid) in those cycles.

A single-place buffer is a corner case of such blocks: Figure 5 shows a single-place buffer that can be used to pipeline long wires since it always takes at least one cycle for a token to move downstream and a stop signal to move upstream. The usual AND gate regulating firing is simplified away by don't-cares.

While unit-rate dataflow blocks are somewhat limited, they form the basis of Carloni et al.'s latency-insensitive design [4]. In LID terminology, our single-place buffer is a "relay station" and our multi-input, multi-output unit-rate blocks are "shells" whose cores are "pearls."

5. Core Logic: General Dataflow Blocks

Unit-rate dataflow blocks are useful, but their "AND" firing rule makes them limited. We designed our buffers to support blocks with general firing rules [11]. For the unit-rate blocks described above, the core is only required to heed the *fire* signal, which abstracts all the details about token and output buffer availability. In general, a core may "peek" at input tokens without consuming them, choose to consume only certain inputs, and choose to emit only certain outputs, all depending on internal state, the presence, and the value of input tokens.

A full discussion of the approaches, considerations, and pitfalls of designing general dataflow blocks and the core

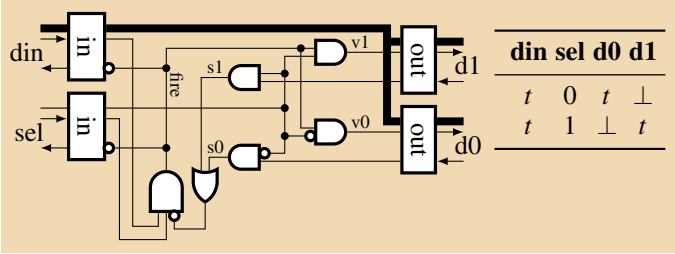


Figure 6. A *switch* block implementation and its firing rules. When tokens are present on both din and sel, the block transfers the token on din to d0 or d1 depending on the value of sel.

```
always @(*) begin
  cdinstop = 1; cselstop = 1;          // Hold inputs
  cd0data = { 1'b0, 8'bx };            // ⊥
  cd1data = { 1'b0, 8'bx };            // ⊥
  if (cdindata[8] && cseldata[1] && // Din, sel = 0?
      !cseldata[0] && !cd0stop ) begin
    cdinstop = 0; cselstop = 0;        // Consume both
    cd0data = cdindata;                // Emit on D0
  end else if (cdindata[8] && cseldata[1] && // din,
               cseldata[0] && !cd1stop) begin//sel=1?
    cdinstop = 0; cselstop = 0;        // Consume both
    cd1data = cdindata;                // Emit on D1
  end
end
```

Figure 7. Verilog for the core of the *switch* block with 8-bit data

logic for them falls well outside the scope of this paper. Below, we give some illustrative examples that are fairly well-behaved, but general blocks present many challenges. For example, it is easy to inadvertently build networks that are sensitive to latency, i.e., whose functional behavior changes dramatically with the addition of seemingly unimportant delays, violating a typical objective of a dataflow methodology.

Provided the core of each dataflow block is finite-state, dataflow systems built using our buffers are also finite-state and thus properties such as determinism and deadlock-freedom can always be tested through exhaustive simulation (e.g., model-checking). In practice, however, such analysis quickly grows impractical for larger systems. An alternative is to follow a discipline that guarantees such emergent properties, such as Kahn's [9], which guarantees I/O properties are delay-invariant provided the system uses only blocking reads on input buffers.

5.1. Switch

As an illustration of how richer blocks can be implemented, consider the demultiplexing *switch* block described by Lee and Parks [11]. This block has two data outputs *d0* and *d1*, one data input *din*, and a Boolean *select* input. The value of the token that arrives on the *select* input directs the next token that arrives on the *din* input to either *d0* or *d1*. Unlike a unit-rate dataflow block, which output emits a token depends on the value of an input.

Figure 6 illustrates how our blocks can be used to implement *switch*. The datapath is simple: the data from *din* is sent to both *d0* and *d1*, but the token is never duplicated because *v0* and *v1* are never true simultaneously. Instead, this block is primarily control logic.

First, if either *din* or *sel* does not have a token, *fire* is false, no input token is consumed, and both *v0* and *v1* are false, so no output tokens are produced.

Now, assume *din* and *sel* both have tokens. Here, *sel* is true if the data is to be sent to *d1*. If the *d1* output buffer is prepared to accept a token, *s1* will be false so *fire* will be true, setting *s1* true and *s0* false. Conversely, if *sel* is false, *s0* will reflect whether *d0* is prepared to accept a token and if so, *fire* will drive *v0* true and *v1* false.

The design of *switch* illustrates the general technique for designing dataflow blocks using our buffers. The control logic is responsible for determining when the block can fire—when the block has an acceptable combination of input tokens and available output buffers. For the *switch* block to fire, the *din* and *sel* inputs and either the *d0* or *d1* output must be available.

Such control logic is easy code in an HDL: by default, no input tokens should be consumed and no output tokens emitted, but if the input tokens' presence and values meet a firing condition, inputs are consumed and outputs produced.

Figure 7 illustrates a natural way to code the core of the *switch* block. By default, all inputs are told to hold any incoming tokens and all outputs are told not to emit anything (i.e., given ⊥ tokens). Then each firing condition is tested, which includes checking for the presence of input tokens and space for output tokens. The first test checks if *din* has a token, *sel* has a 0 token, and output *d0* can accept a token. If so, tokens on *din* and *sel* are consumed and the token on *din* is sent on *d0*. Note that the predicate guarantees all these communication actions can take place. The second rule is similar except it checks for the presence of a 1 token on *sel* and routes the incoming token to *d1*.

5.2. Fork and Eagerness

A fork block has one input and multiple outputs. Each input token is duplicated and sent on every output. A straightforward implementation fires only when the input token is available and every output is ready to accept data. Figure 8 shows code for a three-output fork.

While easily coded, such a strict implementation is conservative about emitting data and may need more buffering to avoid needless deadlocks. Specifically, although outputs could be generated when the input token has arrived and *any* output is free, this implementation waits until *every* output is free.

Our buffers allow the core to "peek" at an input token without consuming it, enabling a less conservative firing policy at the expense of a more complicated core. Once an input token has arrived, the core can start immediately

```
always @(*) begin
   cistop = 1;                          // Hold input
   co1data = {1'b0, 8'bx};              // output1: ⊥
   co2data = {1'b0, 8'bx};              // output2: ⊥
   co3data = {1'b0, 8'bx};              // output3: ⊥
   if (cidata[8] && !co1stop &&         // Input available,
       !co2stop && !co3stop) begin      //  outputs free?
      cistop = 0;                       // Consume input
      co1data = cidata;                 // Send token
      co2data = cidata;                 //  on all three
      co3data = cidata;                 //  outputs
   end
end
```

Figure 8. Verilog for the core of a conservative three-output *fork* block that waits for all outputs to have space before emitting anything

```
reg [3:1] waiting = 3'b111, willWait;
always @(posedge clk) waiting <= willWait;

always @(*) begin
   cistop = 1;                          // Hold the input
   co1data = {1'b0, 8'bx};              // ⊥
   co2data = {1'b0, 8'bx};              // ⊥
   co3data = {1'b0, 8'bx};              // ⊥
   if (cidata[8]) begin                 // Has the input arrived?
      willWait = waiting & {co3stop, co2stop, co1stop};
      if (waiting[1] && !co1stop) co1data = cidata;
      if (waiting[2] && !co2stop) co2data = cidata;
      if (waiting[3] && !co3stop) co3data = cidata;
      if ( !(| willWait) ) begin        // Still waiting?
         cistop = 0;                    // No: consume the input
         willWait = 3'b111;             // Wait next time
      end
   end else willWait = 3'b111;          // Wait next time
end
```

Figure 9. Verilog for the core of an eager three-output *fork* block that starts trying to emit outputs as soon as it has an input

sending it to open outputs, but the core needs to remember to which outputs it has already sent the token to avoid inadvertently sending multiple copies of the token on the same output.

Figure 9 illustrates an eager *fork* block that starts sending the input token as soon as it can but only consumes the token once all the outputs have been sent. The important difference here is that the core needs to track additional state: each bit of *waiting* indicates whether the core is waiting to send a token on that output. Each bit in the *willWait* vector indicates whether the output still needs to wait after the current cycle—when it was waiting and the output is not yet available. When every bit in *willWait* is false, a token has been sent on every output and the token is consumed.

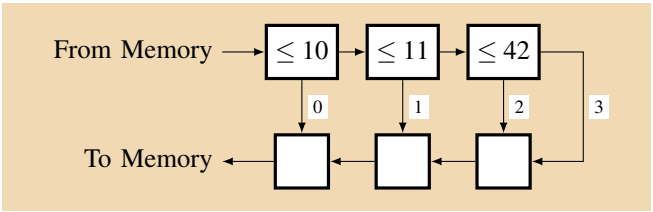


Figure 10. A range partitioning pipeline configured with splitters (10, 11, 42). Data is loaded from memory and flows right until it satisfies the predicate, then it falls, is tagged with the bucket number, and flows left to memory.

6. An Application: Partitioning for Database Accelerators

Our need for efficient, flexible dataflow blocks arose as we were developing a specialized accelerator for database processing [16]. We believe the road to greater silicon efficiency lies in specialization: instead of one-size-fits-all general-purpose processors, the world needs highly efficient application-specific processors that can be powered off when unused. Quickly designing such processors correctly is not easy; composable building blocks such as ours helps to simplify the process.

When manipulating large datasets, partitioning is often critical for bring data sizes under control. Modern databases routinely partition data to make tasks possible and efficient.

The pipeline we present here performs *range partitioning*: it divides integer data into buckets identified by contiguous, non-overlapping ranges whose boundaries are defined by *splitters*. Figure 10 shows an example of our pipeline configured with the three splitters (10, 11, 42). This divides the data into four buckets labeled ≤ 10, $10 < x \leq 11$, $11 < x \leq 42$, and > 42.

Our pipeline consists of two types of blocks. The split blocks—those on the top row—compare the key of each incoming token (e.g., 32 bits of a 256-bit payload) with the block's splitter. If the key is less than or equal to the splitter, the token is sent south and tagged with the number of its bucket. Otherwise, the token is sent east to the next split block. Each block on the bottom row performs a nondeterministic merge of tokens from its two inputs, sending the now-tagged tokens west toward memory. The merge blocks need an arbitration rule for when there are tokens on both inputs; our current block simply prefers its north input over its east and could be modified to be more fair, but the specific policy does not matter for this application provided every input token is routed eventually.

Testing many predicates simultaneously provides the parallelism in this pipeline; keeping the data local improves performance. A lower-latency parallel architecture might distribute data to all predicate tests simultaneously, but such a broadcast would lead to longer wires and lower overall throughput.

The merge blocks makes the token sequence generated by this pipeline potentially nondeterministic, but determinism can be recovered by grouping output tokens by their

```
module split #(parameter TAG = 0, W = 64)
 (input clk,              input [31:0]  splitter ,
  input [W:0]    wdata, output wstop,
  output [W:0]   edata, input estop,
  output [W+8:0] sdata, input sstop );

wire [W:0] cwdata; reg  cwstop;
inbuf #(W)    ib(clk, wdata, wstop, cwdata, cwstop);

reg [W:0] cedata; wire cestop;
outbuf #(W)  o1(clk, cedata, cestop, edata, estop);
reg [W+8:0] csdata; wire csstop;
outbuf #(W+8) o2(clk, csdata, csstop, sdata, sstop);

always @(*) begin
 cwstop = 1;                          // do not consume
 cedata = {1'b0, {W{1'bx}}};          // ⊥
 csdata = {1'b0, 8'bx, {W{1'bx}}};
 if (cwdata[W]) // input token available?
   if (cwdata[31:0] <= splitter ) begin // in range?
     if (!csstop) begin // can we send south?
       csdata = {1'b1, TAG[7:0], cwdata[W-1:0]};//tag
       cwstop = 0; // consume input
     end
   end else if (!cestop) begin // can we send east?
     cedata = cwdata; // send east
     cwstop = 0; // consume input
   end
end
endmodule
```

Figure 11. The split block: if the token from the west input is less than or equal to the splitter, tag the token and send it south, otherwise, send it east.

```
module merge #(parameter W=72)
   (input clk,
    input [W:0] ndata, output nstop,
    input [W:0] edata, output estop,
    output [W:0] wdata, input  wstop);

wire [W:0] cndata; reg  cnstop;
inbuf  #(W) i1(clk, ndata,  nstop,  cndata, cnstop);
wire [W:0] cedata; reg  cestop;
inbuf  #(W) i2(clk, edata,  estop,  cedata, cestop);
reg [W:0] cwdata; wire cwstop;
outbuf #(W) ob(clk, cwdata, cwstop, wdata,  wstop);

always @(*) begin
   cnstop = 1; cestop = 1;       // Do not consume
   cwdata = {1'b0, {W{1'bx}}}; // ⊥
   if (cndata[W] && !cwstop) begin // North token?
     cwdata = cndata; cnstop = 0; // send west
   end else if (cedata[W] && !cwstop) begin // east?
     cwdata = cedata; cestop = 0; // send west
   end
end
endmodule
```

Figure 12. The merge block: if a token is available from the north, pass it west. Otherwise, if a token is available from the east, pass it west.

tags. In our application, we assume data will be read in bursts from memory and clustered back into bursts according to their tags before being sent back to memory.

Figure 11 and Figure 12 show the Verilog for the body of the split and merge blocks. These are simple but were fussy to code. It is important to be disciplined about names and structure. Give each input and output a simple name and prefix it with "c" to denote the signals the combinational core may use. Instantiate each input and output port, connecting each between module ports and the "c" signals for the combinational core. In the combinational core block, set all input *stop* signals to 1 and all output token values to $\perp$ as a default. Then, check input tokens, their values, and finally the availability of output ports before sending data on those outputs and consuming input tokens.

7. Experimental Results

In Verilog, we coded a parametric version of the range paritioner pipeline shown in Figure 10 that uses the split, merge, inbuf, and outbuf code presented in Figures 11, 12, and 2. We synthesized pipelines of various sizes using

TABLE 1. SYNTHESIS RESULTS FOR THE RANGE PARTITIONING PIPELINE

Splitters	Token	F_{max}	Area Resources		
	Bits	MHz	ALMs	%	Registers
2	32	167	189	1	414
2	64	157	350	1	798
2	128	152	672	2	1573
32	128	137	10821	26	25536
64	128	140	21704	52	51168
4	64	158	700	2	1621
8	64	145	1409	3	3261
16	64	147	2826	7	6559
32	64	144	5682	14	13148
64	64	138	11404	27	26414
128	64	140	22914	55	53087

Quartus 14.0 targeting an Altera Cyclone V (5CSXFC6D6F31C8ES) FPGA: 41910 ALMs (Adaptive Logic Modules: 7-input lookup tables).
Bits is the width of each data token (only first 32 are considered)
F_{max} is the maximum operating frequency.
Regs. is the number of registers in the design.
166 MHz target clock rate.

Altera's Quartus and targeted a mid-speed Cyclone V FPGA that is attractive for our application because it also has hard ARM processor cores and DRAM controllers, making it a good vehicle for experimenting with application-specific accelerators. For reference, the (customized) memory and multiplier blocks on these chips can run at about 300 MHz.

Table 1 shows the results of synthesizing pipelines of various depths and numbers of bits. F_{max} is the main figure

of merit: the highest frequency at which the FPGA can run under worst-case conditions (85C, 1.1V). The "ALMs" column gives a rough measure of area: a count of how many of Altera's "Adaptive Logic Modules," primarily 8-input lookup tables that form the bulk of the FPGA fabric, were used. This particular chip had 41910 available ALMs; the percentage number in the next column lists the fraction of available ALMs consumed by our design. Finally, the "Regs" column lists the number of registers (flip-flops) in the design: each ALM has four available.

The operating frequency of our pipelines are largely independent of their size, as expected. At most, we see a 14% drop (158 MHz to 140) over a 32× increase in pipeline size (4 to 128 splitters with 64-bit tokens). We suspect this may be due to the synthesis tool, or perhaps its concern about clock skew as the design swells from a minuscule fraction of the chip to over half. The effect is smaller for a comparable increase in a wider pipeline (2 to 64 splitters with 128-bit tokens). That the frequency does not change monotonically with pipeline size also suggests a certain level of randomness in the synthesis tool. The resource usage is also predictably linear.

8. Related Work

Dataflow has been implemented in hardware for a long time, but we were particularly inspired by the latency-insensitive design technique of Carloni et al. [4], which advocates partitioning systems into blocks that communicate exclusively through buffers to eliminate long wires. The LID approach, however, starts from a synchronous design that implicitly assumes each wire always communicates one bit per clock cycle. While this enables an existing design to be made latency-insensitive, it often leads to unnecessary communication and lower performance. It is possible to heed don't-care information and complete computation before unneeded tokens are available [12], but those tokens must still be transmitted. While we adopt the buffering methodology of LID and one of its protocols [13], we expose the notion of tokens and backpressure to the designer, raising the level of abstraction and enabling better performance.

Cortadella et al.'s Synchronous Elastic Architectures [5] combine ideas from LID and asynchronous logic. They propose a lightweight latch-based approach for introducing latency, but, like LID, effectively only support unit-rate dataflow (i.e., that can be modeled with marked graphs) and thus cannot describe the richer dataflow blocks that we can. Their latch-based approach could probably be applied in our setting, but we have not attempted to do so, in part because we have been targeting FPGAs.

Janneck et al. [6, 7, 1] propose the CAL language for describing dataflow blocks and a system for synthesizing it into hardware. Siret et al. [14] take a similar approach. Like us, they implement robust, high-speed dataflow networks in hardware. However, they use relatively heavyweight FIFOs for inter-block communication. We were unable to obtain their tools to perform a head-to-head comparison of their protocols and buffers, but we imagine our work might integrate nicely with theirs.

Williamson and Lee [15] consider synthesizing SDF graphs which, like the unit-rate graphs of LID, cannot produce or consume tokens based on data. They also gave themselves the bigger challenge of sharing physical blocks across multiple logical operations in a source graph, which we do not consider.

Jung et al. [8] also synthesize SDF graphs into hardware, which allows them to know the schedule of operations at compile time and consider things like performing multiple operations in a single clock cycle. Our technique is much more coarse-grain, but allows data-dependent operation, which it outside the scope of SDF.

9. Conclusions and Future Work

We presented a technique for building dataflow networks in hardware that allows a designer to express the function of each block just in terms of the production of output tokens and the consumption of input tokens, something that be merely a combinational function. Adding our buffers to the inputs and outputs of such blocks eliminates combinational paths between ports, allowing the blocks to be chained together into arbitrarily large networks without reducing the clock rate. We gave proofs that our input and output blocks deliver data reliably without duplicating or dropping tokens.

We showed how to use our buffers to implement pipelined communication channels, unit-rate dataflow blocks (e.g., like those in latency insensitive design), and richer dataflow blocks that consider data values when deciding which tokens to consume and emit.

The need for specialized hardware to accelerate database applications was the original motivation for this work. We showed how our technique could be used to produce a high-speed pipeline for performing range partitioning—a common core operation in databases. We implemented this pipeline on an FPGA and showed that its maximum clock rate dropped only slightly as the pipeline grew larger.

For the sake of exposition, our range partitioning pipeline is substantially simplified from one we eventually plan to use in practice. One issue is completion detection: this pipeline is intended to be used on multiple tables, each with their own splitters. It will be necessary to know when all tokens have trickled out of the pipeline and written back to memory so the splitters can be change and the next partitioning operation started. We envision sending a sentinel token that is copied to both outputs of each split block and merged back into a single token by each merge block.

We also did not address how the splitters are to be loaded into the pipeline. Because this configuration data is only loaded per table instead of per token, a shift register without flow control would suffice.

Another realistic extension would be to adapt the merge blocks to handle memory write bursts. Tokens would accumulate on the north input of each merge block until there were enough for a DRAM write burst, then the entire burst

would proceed through the pipeline to the memory and be written as a single unit.

While range partitioning is a useful, common operation, our pipeline structure is easily augmented to perform other useful operations. Expanding the range of operations the pipeline should perform without it growing slow and overly complicated is ongoing work.

Our approach is not directly applicable to the implementation of multi-rate dataflow networks such as SDF [10] because our buffers only consider transferring single tokens, but it is well-suited to the periodic binary firing rules of cyclo-static dataflow [2], a technique familiar in hardware dataflow implementations [3].

Going forward, we envision a tool that takes high-level specifications of dataflow blocks and produces circuits like Janneck et al. [7]. Of course, we would use our buffers, but we also want to consider when they can be avoided. Clustering dataflow blocks into larger regions that operate in a single cycle could produce more efficient designs, provided the blocks do not grow big enough to impact the maximum clock rate. We plan to develop a clustering algorithm.

References

[1] Endri Bezati, Marco Mattavelli, and Jörn W. Janneck. High-level synthesis of dataflow programs for signal processing systems. In *Proceedings of the International Symposium on Image and Signal Processing and Analysis (ISPA)*, pages 750–755, Trieste, Italy, September 2013.

[2] Greet Bilsen, Marc Engels, Rudy Lauwereins, and J. A. Peperstraete. Cycle-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, February 1996.

[3] Joseph Buck and Radha Vaidyanathan. Heterogeneous modeling and simulation of embedded systems in El Greco. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign (CODES)*, San Diego, California, May 2000.

[4] Luca. P. Carloni, Kennneth L. McMillan, Alexander Saldanha, and Alberto L. Sangiovanni-Vincentelli. A methodology for correct-by-construction latency-insensitive design. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 309–315, San Jose, California, November 1999.

[5] J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of synchronous elastic architectures. In *Proceedings of the 43rd Design Automation Conference*, pages 657–662, San Francisco, California, July 2006.

[6] Johan Eker and Jörn W. Janneck. CAL language report: Specification of the CAL actor language. Technical Report UCB/ERL M03/48, EECS Department, University of California, Berkeley, December 2003.

[7] Jörn W. Janneck, Ian D. Miller, David B. Parlour, Ghislain Roquier, and Matthieu Wipliez Mickaël Raulet. Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study. *Journal of Signal Processing Systems*, July 2009. Online.

[8] Hyunuk Jung, Kangnyoung Lee, and Soonhoi Ha. Efficient hardware controller synthesis for synchronous dataflow graph in system level design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(4):423–428, August 2002.

[9] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of IFIP Congress 74*, pages 471–475, Stockholm, Sweden, August 1974. North-Holland.

[10] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[11] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.

[12] Cheng-Hong Li and Luca P. Carloni. Leveraging local intra-core information to increase global performance in block-based design of systems-on-chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(9):165–178, February 2009.

[13] Cheng-Hong Li, Rebecca Collins, Sampada Sonalkar, and Luca P. Carloni. Design, implementation, and validation of a new class of interface circuits for latency-insensitive design. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 13–22, Nice, France, May 2007.

[14] Nicolas Siret, Watthieu Wipliez, Jean-François Nezan, and Aimad Rhatay. Hardware code generation from dataflow programs. In *Proceedings of Design and Architectures for Signal and Image Processing (DASIP)*, pages 113–120, Edinburgh, Scotland, October 2010.

[15] Michael C. Williamson and Edward A. Lee. Synthesis of parallel hardware implementations from synchronous dataflow graph specifications. In *Proceedings of the Asilomar Conference on Signals, Systems & Computers*, pages 1340–1343 vol. 2, Pacific Grove, California, November 1996.

[16] Lisa Wu, Orestis Polychroniou, Raymond J. Barker, Martha A. Kim, and Kenneth A. Ross. Energy analysis of hardware and software range partitioning. *ACM Transactions on Computing Systems*, 32(3):8, August 2014. 24 pages.