Compositional Dataflow Circuits

STEPHEN A. EDWARDS, RICHARD TOWNSEND, MARTHA BARKER, and MARTHA A. KIM, Columbia University

We present a technique for implementing dataflow networks as compositional hardware circuits. We first define an abstract dataflow model with unbounded buffers that supports data-dependent blocks (mux, demux, and nondeterministic merge); we then show how to faithfully implement such networks with bounded buffers and handshaking.

Handshaking admits compositionality: our circuits can be connected with or without buffers, and combinational cycles arise only from a completely unbuffered cycle. While bounding buffer sizes can cause the system to deadlock prematurely, the system is guaranteed to produce the same, correct, data before then. Thus, unless the system deadlocks, inserting or removing buffers only affects its performance. We demonstrate how this enables design space to be explored.

CCS Concepts: • **Theory of computation** → *Streaming models*; • **Hardware** → *Hardware description languages and compilation*; • **Computing methodologies** → Parallel programming languages;

Additional Key Words and Phrases: Kahn networks, high-level synthesis, dataflow

ACM Reference format:
Stephen A. Edwards, Richard Townsend, Martha Barker, and Martha A. Kim. 2019. Compositional Dataflow Circuits. *ACM Trans. Embed. Comput. Syst.* 18, 1, Article 5 (January 2019), 27 pages.
https://doi.org/10.1145/3274280

1 INTRODUCTION

Dataflow networks are a natural model for parallel, distributed computation [13, 27, 29, 30]. Processes in a network execute in parallel and communicate via sequences of tokens passed over unbounded channels. These networks are well-suited to specifying complex hardware designs because of their "patience": process speed has no effect on network function. We address the challenge of implementing such a network with fast, correct hardware.

We describe and experimentally validate a technique for synthesizing synchronous digital circuits that implement a restricted class of Kahn process (dataflow) networks [27]. Our approach is compositional: each Kahn process becomes a circuit that may be connected to others with or without buffering, making it easy to consider a variety of designs. For example, buffer-free connections are fast but lead to combinational paths that limit clock speed; inserting buffers breaks long paths at the expense of latency.

Our generated circuits retain the "patience" of Kahn's formalism through a valid/ready flow-control protocol (i.e., backpressure); local handshaking eliminates any global controller (and thus

Authors' addresses: S. A. Edwards, R. Townsend, M. Barker, and M. A. Kim, Department of Computer Science, Columbia University, 500 W 120 St., Room 450, New York, NY 10027; emails: {sedwards, rtownsend, mbarker, martha}@cs.columbia.edu.

1539-9087/2019/01-ART5 $15.00
https://doi.org/10.1145/3274280

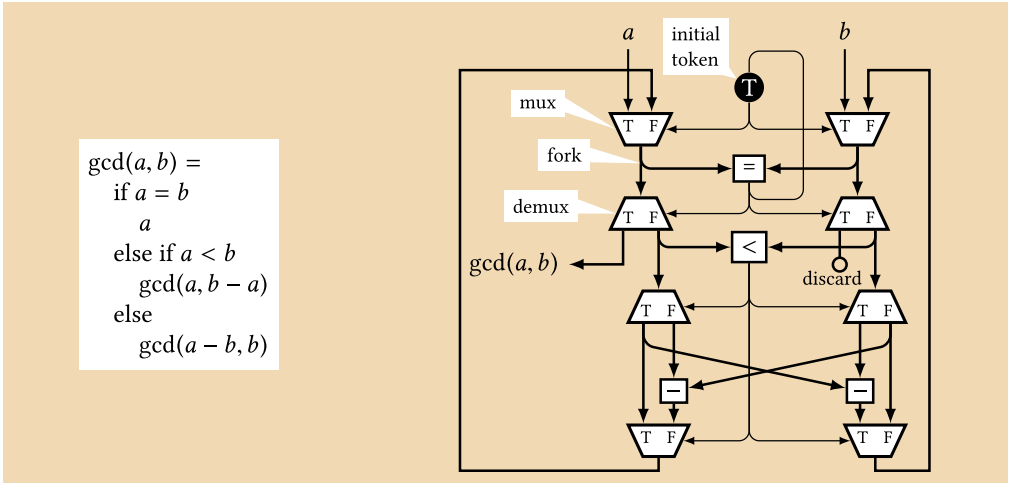


Fig. 1. A recursive definition of Euclid's greatest common divisor algorithm and a dataflow network implementing it. The tail recursion is implemented with feedback loops.

long signal lines) and enables the insertion and removal of buffering. This accommodates blocks with dynamically varying latency (e.g., memory controllers) and makes it easy to adjust the number of pipeline stages, even in the presence of feedback.

We use bounded buffers, so our networks may deadlock when a network with unbounded buffers could continue. This only truncates the sequences of data passed between processes; it does not change their values. Pingali and Arvind [35] show this. To impose demand-driven scheduling on a Kahn network, they introduce *demand streams* that run opposite each communication channel. Such streams model buffer capacity: each process is forced to consume (and thus potentially wait for) an incoming demand token before it sends a token on the corresponding output channel. Similarly, after a process consumes a normal input token, it immediately emits a corresponding demand token. Pingali and Arvind show that a network augmented with such streams produces on each channel a prefix of the stream of tokens that would be produced in the original network. Geilen and Basten [18] present an alternative proof.

Choosing appropriate buffer sizes can range from straightforward to undecidable. By design, a dataflow network generated from a syntax-directed translation of a structured program (e.g., Dennis [13]) never requires the accumulation of an unbounded number of tokens to run, so inserting buffers according to a simple policy suffices to prevent these networks from deadlocking. We use such a policy for the networks we generate from Haskell programs [43], the original motivation for our work, but such a technique often inserts more buffers and hence more latency than strictly necessary. Our complete translation of Haskell assumes a sufficiently large heap, since the Haskell programs we translate can require unbounded memory to run to completion.

Unfortunately, the optimal buffer insertion problem for a dataflow network with arbitrary topology is undecidable: Buck [5] showed that a very simple subset of Kahn processes (such as ours, which include data-dependent mux and demux) is Turing complete, rendering undecidable the question of whether arbitrary networks of our actors and buffers can run without deadlocking. As such, we do not propose a buffer sizing algorithm here, but note numerous authors have proposed approaches that may be adaptable to our networks [2, 17–21, 33, 34, 41].

Figure 1 illustrates a dataflow network we can implement. This uses Euclid's algorithm to compute the greatest common divisor of pairs of tokens arriving on the two input channels. For

example, if channel a receives tokens 100 and 56 and channel b receives 45, 49, and 3, then the output will be $5 = \gcd(100, 45)$ followed by $7 = \gcd(56, 49)$. The 3 on b is ignored, because no mating token ever arrives on a.

In this network, an initial "T" token is fed to the top row of multiplexers, instructing each to steer a token from an input to the equality comparator actor ("="). Because the channels from the multiplexers fork, the first row of demultiplexers also receive copies of these tokens. If the tokens are equal (the base case for the algorithm), then the comparator emits a T, causing the demultiplexers to emit the a token as the result and discard the b token. The T from the comparator is also fed back to both input multiplexers, prompting them to accept a new pair of tokens from the inputs.

In the recursive case, the tokens differ, prompting the demultiplexers to send copies of the tokens to the magnitude comparator (<) and to the second row of demultiplexers. The output of the magnitude comparator flows to the second demuxes and bottom multiplexers, which together control whether a is subtracted from b or b is subtracted from a. Since the equality comparator emitted an F, the outputs from the bottom multiplexers are fed around and flow back through the top multiplexers and the process repeats.

We synthesize networks by transforming each actor into a small block of logic and replacing each channel with a mixture of point-to-point connections, buffers, and fork circuitry. Figure 16 shows our preferred buffering of this GCD network.

We have implemented a simple compiler that transforms a textual description of our networks into synthesizable SystemVerilog. Figure 11 shows how we express a fragment of the GCD network in our textual language.

We make the following contributions:

- circuits for a small, rich family of data-dependent dataflow actors, which can be composed without buffering yet are safe from spurious combinational cycles (Section 3);
- a novel way to implement a nondeterministic merge that reports its choices, allowing it to safely manage shared resources (Section 3.4);
- a novel approach to breaking long combinational paths and loops that uses two distinct types of buffers: one for the data network and one for backpressure (Section 4);
- a typed "assembly language" for describing our networks with polymorphic actors and algebraic data types that we can compile into synthesizable SystemVerilog (Section 6);
- experiments that show how buffering may be added to explore the design space without changing functionality (Section 7).

In the next section, before we present our circuits, we formally define the semantics of our unbounded dataflow networks.

2 SPECIFICATIONS: KAHN NETWORKS

Our goal is a hardware implementation of a dataflow network. Here, we describe our specifications: a restricted class of Kahn networks with unbounded buffers. Our specifications, expressed in this model of computation, are deliberately more abstract than our implementations to allow buffers to be added and removed (e.g., to adjust pipeline depth) as part of the implementation process.

This section is largely review: Kahn [27] provides the framework, Lee and Matsikoudis [29] show how to model firing rules, and our model of nondeterministic merge is due to Broy [4].

2.1 Kahn Networks

A Kahn network consists of Kahn processes that pass around tokens. Kahn networks are deterministic, because the processes are continuous, meaning that supplying additional input tokens

can only produce additional output tokens; a Kahn process cannot "change its mind" once it has decided to emit a token. Below, we formalize such networks.

A Kahn network passes around *tokens* drawn from a set Σ. This set is typically finite and often includes 32-bit binary integers, but its structure is irrelevant for the semantics we present in this section; see Section 6 for how we construct sets of tokens in practice. Our networks do not necessarily terminate, so we consider both finite and infinite sequences of tokens flowing on channels and write $S = \Sigma^* \cup \Sigma^\omega$ for the set of such sequences. Note that the empty sequence ϵ is included in this set ($\epsilon \in \Sigma^*$). Juxtaposition will denote concatenation, e.g., for two tokens $x, y \in \Sigma$, xy represents the two-token sequence consisting of x followed by y. We also use juxtaposition for concatenation of sequences: if $a \in \Sigma^*$ is a finite sequence and $b \in S$, then ab is the sequence a followed by b.

We use prefix ordering on sequences. We write $a \sqsubseteq b$ if a is a prefix of b or a is equal to b. It follows that $\sqsubseteq$ is a partial order. Technically $a \sqsubseteq b$ iff $a = b$ or $\exists c \in S$ s.t. $ac = b$. We extend this ordering elementwise to n-tuples of sequences (written in bold): if $a_1, \ldots, a_n, b_1, \ldots, b_n \in S$, $\mathbf{a} = (a_1, \ldots, a_n) \in S^n$, and $\mathbf{b} = (b_1, \ldots, b_n) \in S^n$, then we write $\mathbf{a} \sqsubseteq \mathbf{b}$ iff $a_1 \sqsubseteq b_1$, $\ldots$, $a_n \sqsubseteq b_n$. Juxtaposition of n-tuples of sequences denotes elementwise concatenation: $\mathbf{ab} = (a_1 b_1, \ldots, a_n b_n)$.

A *Kahn process* is a continuous function $P : S^n \to S^m$ that takes a tuple of n input sequences and produces a tuple of m output sequences. Continuity means P is monotonic, so $\mathbf{a} \sqsubseteq \mathbf{b}$ implies $P(\mathbf{a}) \sqsubseteq P(\mathbf{b})$. Equivalently, providing P with additional tokens may produce more output tokens, but tokens that have already been produced cannot be changed or rescinded. Continuity also means a process cannot produce an output only after an infinite time. See Lee and Matsikoudis [29] for a formal discussion of continuity.

As an example, consider a process that adds two input sequences to produce an output sequence. Assume integer-valued tokens, i.e., $\Sigma = \mathbb{Z}$. This process computes the pairwise sum of the two sequences up to the end of the shorter sequence, i.e.,

$$P(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m) = w_1 w_2 \cdots w_{\min(m,n)}, \tag{1}$$

where $w_i = x_i + y_i$ and sequence lengths m and n may be zero, finite, or infinite.

A Kahn network is a collection of Kahn processes whose input sequences are supplied through channels, each of which is either supplied by the environment or the output of some process. A *Kahn network* $N = (\mathbf{P}, e, M)$ is a triple consisting of a vector of r Kahn processes $\mathbf{P} = \{P_1, \ldots, P_r\}$, a number $e \in \{0, 1, \ldots\}$ of input channels from the environment, and a "wiring matrix" function $M : \{1, \ldots, r\} \times \{1, \ldots\} \to \{1, \ldots, e\} \cup (\{1, \ldots, r\} \times \{1, \ldots\})$ that maps each process input (a process number and input index) to either one of the e environment channels or the output of some process. The M function is pure bookkeeping: it merely encodes the connectivity of the network (i.e., a graph) by specifying the source of each input to each process.

Let $c_{i,j}$ be output j from process i, c_k be environmental channel k, m_i be the number of outputs from process i, and let

$$\mathbf{c} = (\underbrace{c_1, \ldots, c_e}_{e \text{ inputs}}, \underbrace{c_{1,1}, \ldots, c_{1,m_1}}_{\text{process 1 outputs}}, \underbrace{c_{2,1}, \ldots, c_{2,m_2}}_{\text{process 2 outputs}}, \ldots, \underbrace{c_{r,1}, \ldots, c_{r,m_r}}_{\text{process } r \text{ outputs}})$$

be the vector of all channels in the system. The *behavior* of a Kahn network for input $(c_1, \ldots, c_e)$ is the least $\mathbf{c}$ satisfying

$$\begin{aligned} (c_{1,1}, \ldots, c_{1,m_1}) &= P_1(c_{M(1,1)}, \ldots, c_{M(1,n_1)}) \\ &\vdots \\ (c_{r,1}, \ldots, c_{r,m_r}) &= P_r(c_{M(r,1)}, \ldots, c_{M(r,n_r)}), \end{aligned} \tag{2}$$

where n_k is the number of inputs on the kth process and $c_{M(k,l)}$ is the channel feeding the lth input of the kth process: either an environment channel (i.e., $1 \le M(k,l) \le e$) or a specific output channel of a specific process (i.e., $M(k,l) = i,j$, where $k, i \in \{1 \ldots r\}$, $1 \le l \le n_k$, and $1 \le j \le m_i$).

Channels may "fork": each channel has a single source (either a process output or the environment) but may have multiple receivers. I.e., $M(k_1,l_1) = M(k_2,l_2)$ may hold for some $(k_1,l_1) \neq (k_2,l_2)$.

Kahn showed [27] that his networks are *deterministic*: there is exactly one least $\mathbf{c}$ that satisfies Equation (2) for each tuple of input sequences provided the P_i are continuous.

2.2 Dataflow Actors

The Kahn formalism describes our networks; we follow Lee and Matsikoudis's formalism for actors [29] for describing processes. Actors react to input tokens according to firing rules; a rule is a tuple of empty or singleton token sequences. When an actor's input matches a rule, the actor consumes the matched tokens and produces a single token on certain outputs. Lee and Matsikoudis use sequences in their firing rules and reactions; we use only singletons, because we target hardware.

Formally, an n-input, m-output *dataflow actor* is a pair (R, f), where $R \subset (\Sigma \cup \epsilon)^n$ are the *firing rules*, $f : R \to (\Sigma \cup \epsilon)^m$ is the *firing function*, and for any $\mathbf{a}, \mathbf{b} \in R$ with $\mathbf{a} \neq \mathbf{b}$ there is no $\mathbf{c}$ such that $\mathbf{a} \sqsubseteq \mathbf{c}$ and $\mathbf{b} \sqsubseteq \mathbf{c}$. This "no-common-prefix" constraint on R ensures the actor behaves as a continuous process: in particular, once an actor can fire on a given rule it cannot fire on another, even if additional tokens arrive.

The process for an actor simply fires repeatedly according to its firing rules. Formally, the Kahn process P for the dataflow actor (R, f) is

$$P(\mathbf{s}) = \begin{cases} f(\mathbf{r})P(\mathbf{t}) & \text{when } \exists \mathbf{r} \in R \text{ such that } \mathbf{s} = \mathbf{rt}; \\ \epsilon^m & \text{otherwise,} \end{cases} \tag{3}$$

where ϵ^m is the m-tuple of empty sequences and juxtaposition represents the pointwise concatenation of sequences. Lee and Matsikoudis [29] showed that P is a continuous function (and thus a Kahn process) provided the firing rules R obey the no-common-prefix rule described above. Note that Equation (3) matches the usual recursive definition of the *map* function familiar to functional programmers.

Our networks allow channels with "initial tokens" to break deadlocks in loops. For example, the top multiplexers in Figure 1 would deadlock without the initial token provided. We model such tokens by allowing processes to emit initial tokens before entering their periodic firing behavior. A *dataflow actor with initial output* is a triple $(R, f, \mathbf{i})$ where R and f are as before and $\mathbf{i} : (\Sigma^*)^m$ is the initial output from the actor. The Kahn process for such an actor is

$$P'(\mathbf{s}) = \mathbf{i}P(\mathbf{s}). \tag{4}$$

2.3 Unit-rate, Mux, and Demux Actors

We construct our networks from three stateless actors. The first, a *unit-rate* actor, waits for a single token on each of its inputs before producing a single output token on each of its outputs.

For example, a two-input process that adds its two integer token inputs is a unit-rate actor. Again, let $\Sigma = \mathbb{Z}$. The actor (R, f) has

$$\begin{aligned} R &= \{(x,y) : x, y \in \mathbb{Z}\} \\ f\Big((x,y)\Big) &= (x+y), \end{aligned} \tag{5}$$

i.e., the actor can fire on any pair of integer tokens (R) and, given such a pair of tokens x and y, the actor produces a single token whose value is $x + y$ (f). It is easy to show that this R follows the no-common-prefix rule. Furthermore, an inductive argument shows that applying Equation (3) to

the R and f in Equation (5) gives the P function in Equation (1). In Figure 1, the equality tester (=), less-than (<), and subtractor (−) actors are each unit-rate.

Our second building block is the *mux* actor (Figure 1 uses four), which consumes a token from its control input to determine from which of its inputs to consume a further token that it then emits on its output channel. For example, a two-way mux actor that takes a 0 or a 1 on its select input has

$$\begin{aligned} R &= \{(0, x, \epsilon) : x \in \Sigma\} \cup \{(1, \epsilon, y) : y \in \Sigma\} \\ f\Big((0, x, \epsilon)\Big) &= (x) \\ f\Big((1, \epsilon, y)\Big) &= (y). \end{aligned} \tag{6}$$

Our third fundamental type of actor is the *demux* (Figure 1 uses four): each input token is routed to an output channel based on a select input token. For a two-output demux,

$$\begin{aligned} R &= \Big\{(x, y) : x \in \{0, 1\}, y \in \Sigma\Big\} \\ f\Big((0, y)\Big) &= (y, \epsilon) \\ f\Big((1, y)\Big) &= (\epsilon, y). \end{aligned} \tag{7}$$

2.4 Nondeterministic Merge

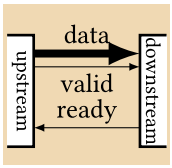
A nondeterministic merge process produces its output sequence by interleaving two or more input sequences. That is, each token of each input sequence appears exactly once and in order in the output sequence, but successive tokens from an input sequence are not necessarily successive in the output. Practically, nondeterministic merge processes expose the timing of a dataflow system implementation by interleaving sequences according to *when* tokens arrive at their inputs, and thus are used to improve performance or break a deadlock by avoiding the need to wait. For example, in the dataflow networks we obtain from syntax-directed translation [43], we use nondeterministic merge processes to share resources as shown in Figure 6, which we will explain in more detail in Section 3.4.

By this definition, a nondeterministic merge process is not a Kahn process (it does not compute a function), so to introduce them into our framework we use a mathematical trick due to Broy [4]: each nondeterministic merge process is one of many different Kahn processes, one for each possible interleaving. The behavior of a system, then, is the set of behaviors produced by the system under every choice of interleaving.

Equivalently, each nondeterministic merge process can be thought of as just a (deterministic) mux actor whose control input is fed from a nondeterministic environment that directs the interleaving of the mux's data inputs. One way to implement a nondeterministic merge process is to have, say, an arbiter decide how to interleave input sequences, and treat the decisions of that arbiter as the control "input" from the environment. While we provide such merge processes, we also provide a merge process with an explicit control stream generated not by the environment, but by the merge process itself. Knowing how a nondeterministic merge interleaved streams is helpful for knowing how to later deinterleave them, which we discuss more in Section 3.4.

3 HARDWARE DATAFLOW ACTORS

In this section and the next, we present our main contribution: a technique for implementing our restricted class of dataflow networks in hardware. Our high-performance circuits facilitate design space exploration, because inserting or removing buffering does not affect what is computed, but removing buffering can introduce deadlock. Multiple actors may be chained together directly to avoid latency (with the danger of increasing the critical path) or buffers may be added to break frequency-robbing critical paths by imposing pipelining.



valid	ready	Meaning
1	0	Token valid; not consumed
1	1	Token transferred
0	–	No token to transfer

Fig. 2. A point-to-point link and its protocol [6]: *valid* indicates the data lines carry a token; *ready* indicates downstream is willing to consume a token.

To implement a dataflow network, each dataflow actor becomes a block of logic with handshaking communication ports, one for each input and output. Each channel in the network becomes a small communication network of wires potentially augmented with fork and buffering circuity (Section 4). Aside from the need to buffer each cycle in the network, the user is free to prescribe buffering to modify the frequency, area, and latency of the synthesized circuit.

In general, the datapath of each actor implements the firing function f of each actor and the flow-control logic implements the firing rules R. Although we do not have formal proofs that show each circuit faithfully implements its specification, we argue for the correctness of our circuits in Section 5.

We present a limited, core group of actors that we have found is rich enough to implement a wide variety of algorithms (see Townsend et al. [43]). Our framework could support additional actor types, but their design is outside the scope of this article. In general, actors should follow the Kahn rule of blocking on exactly one input at a time to avoid nondeterministic behavior.

3.1 Communication and Combinational Cycles

Actors, buffers, and forks in our implementation communicate through unidirectional point-to-point links. We use the bundled-data protocol with handshaking inspired by Carloni's LID [7] and elastic circuits [9] shown in Figure 2. This is a bundled data protocol in which the *valid* bit indicates a token is present on the *data* wires. The downstream block sends the *ready* signal upstream to indicate it is able to consume a token being proffered by the upstream block.

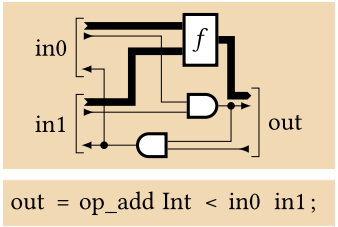
We chose this protocol because it is fast (able to transfer one token per clock cycle indefinitely), patient (both transmitter and receiver can wait indefinitely with no loss of data), and simple (to reduce overhead). We are unaware of other protocols that meet these criteria.

A token is transferred from the upstream block to the downstream block in each cycle in which both *valid* and *ready* are asserted.

We provide a fragment of synthesizable RTL SystemVerilog for each block. To represent, say, an 8-bit channel *c*, we use a nine-bit vector *c* for data (c[8:1]) and *valid* (c[0]), and a wire named *c_r* for *ready*. In our schematics, we label all wires of this port just with the name *c*.

This seemingly simple protocol poses a potentially perilous problem: combinational cycles inadvertently induced by the *ready* signals, which flow backwards through the network. For example, it would be easy to produce a cycle if a *valid* signal depended instantaneously on a *ready* at an output port while a *ready* instantaneously depended on a *valid* at an input port.

We avoid combinational cycles by insisting each cycle in the dataflow network have at least one data and one control buffer (see Section 4) and by insisting no block has a combinational path from a *ready* to a *valid* signal. The data buffer rule eliminates combinational cycles in the data/valid network; the control buffer rule similarly breaks cycles in the *ready* network; and prohibiting combinational paths from *ready* to *valid* means no combinational cycle can include a signal that crosses between the two networks. Intuitively, these rules mean the flow-control network can be scheduled statically: the *valid* network can be computed first (it is acyclic, with inputs from data




```
out = op_add Int < in0 in1;
```

```
assign out =
  { f(in0[W:1], in1[W:1]),          // f(in0, in1)
    in0[0] && in1[0] };             // out valid

assign in0_r = out_r && out[0];     // in0, 1 ready =
assign in1_r = in0_r;               // out valid and ready
```

Fig. 3. A unit-rate actor that computes a combinational function f of two W-bit inputs (bit 0 of each port carries the "valid" flag) once both arrive. Depicted is the circuit, a sample DF specification of the actor, and the corresponding SystemVerilog.

buffers and the environment) followed by the *ready* network, which may take inputs from the *valid* network, control buffers, and the environment.

3.2 Unit-Rate Actors

Figure 3 shows how we implement single-output unit-rate actors such as a two-input adder. First, note that the datapath of this actor is just the combinational function f; our technique merely adds two AND gates for flow-control to the *valid* and *ready* networks. The flow-control logic waits for a valid token on both inputs before asserting the output is valid. The actor indicates it is willing to consume both its inputs when they are both valid and the downstream is also ready to consume the output token. Additional inputs can be added to the circuit of Figure 3 by widening the AND gate for the *valid*s; the *ready* logic remains the same but fans out more widely. An n-output actor can be implemented by n single-output actors with their inputs connected in parallel, although doing so requires *fork* nodes feeding the input channels.

3.3 Mux and Demux Actors

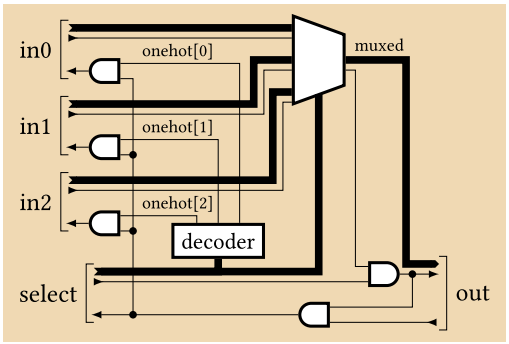
Mux and demux are not unit-rate actors, because they use the value of a select token to determine on which input or output to communicate. A mux uses the value of a selection token to route a token on one selected input to the output. The select token and the token on the selected input must be valid to produce a valid output; input and select tokens are consumed when the output is ready.

The demux is complementary: it directs an input token to a single output depending on the value of a select token. Both the select and input tokens must be valid before a token is proffered on the selected output; that output must be ready before the two tokens are consumed.

Figure 4 shows our three-input multiplexer that routes W-bit tokens. The datapath is a multiplexer that routes one of the inputs to the output depending on the value of the *select* input. A standard one-hot decoder also takes the *select* input and transforms it into a vector that indicates which input should be consumed. We implement the decoder (along with the multiplexer) in the "case" block of the SystemVerilog code: when the *select* value is 0, 1, or 2, the decoder sends 001, 010, or 100 (3'd1, 3'd2, 3'd4), respectively, to the *onehot* vector to select one of the inputs.

The output *valid* bit is the AND of the *valid* from the selected input and the *valid* bit of *select*. *Select* and the selected input are ready when the output is valid and ready.

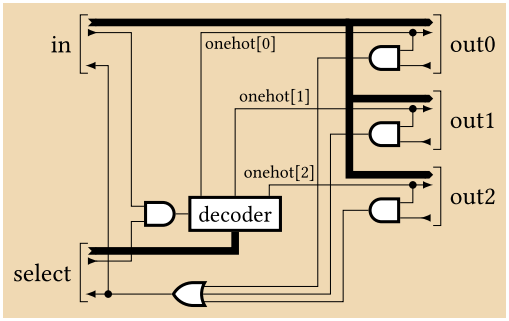
Figure 5 shows a three-output demultiplexer. The datapath is simply fanout that sends the input to all outputs. If both the *in* port and the *select* port have valid tokens, then the one-hot decoder uses the value of the *select* token to indicate which one of the output ports is given a valid token. Both *in* and *select* tokens are consumed if that selected output is ready.




```
out = mux Tri Int < select in0 in1 in2;
```

```
logic [2:0] onehot; // One per input
logic [W:0] muxed;
always_comb
 unique case ( select [2:1])
   2'd0    : {onehot, muxed} = {3'd1, in0};
   2'd1    : {onehot, muxed} = {3'd2, in1};
   2'd2    : {onehot, muxed} = {3'd4, in2};
  default: {onehot, muxed} =
                {3'bx, {{W{1'bx}}, 1'd0}};
 endcase
assign out =
   { muxed[W:1], muxed[0] && select[0] };
assign select_r = out[0] && out_r;
assign {in2_r, in1_r, in0_r} =
                select_r ? onehot : 3'd0;
```

Fig. 4. A three-input W-bit multiplexer; select is 2 bits.




```
out0 out1 out2 = demux Tri Int < select in;
```

```
logic [2:0] onehot;   // One per output
always_comb
   if ( select [0] && in[0]) // Inputs valid?
     unique case ( select [2:1])
       2'd0     : onehot = 3'd1;
       2'd1     : onehot = 3'd2;
       2'd2     : onehot = 3'd4;
      default : onehot = 3'bx;
     endcase
   else onehot = 3'd0;
assign out0 = {in[W:1], onehot [0]};
assign out1 = {in[W:1], onehot [1]};
assign out2 = {in[W:1], onehot [2]};
assign select_r =
  | (onehot & {out2_r, out1_r, out0_r});
assign in_r = select_r ;
```

Fig. 5. A demultiplexer with a two-bit select input and three W-bit outputs.

3.4 Merge Actors

Our implementation of the nondeterministic merge actor is novel: as mentioned in Section 2.4, it is essentially a mux actor whose select "input" is electrically an output. In our formalism, a merge actor is a mux with a nondeterministic select input; in our implementation, the *merge actor itself* generates the tokens on the select channel rather than receiving them.

Figure 6 illustrates how we use our merge node to share a stateless block or subnetwork f that produces one output token per input. The merge actor nondeterministically chooses a token from one of its three inputs to route to the shared f and reports its choice in the form of a token on the select channel. When f produces its result, the demux routes the result to the output corresponding to the chosen input. By using our merge node in this fashion, we enable dynamic load balancing across the users of a shared resource, and simplify the mapping of recursive design specifications onto our networks (see Townsend et al. [43] for further details).

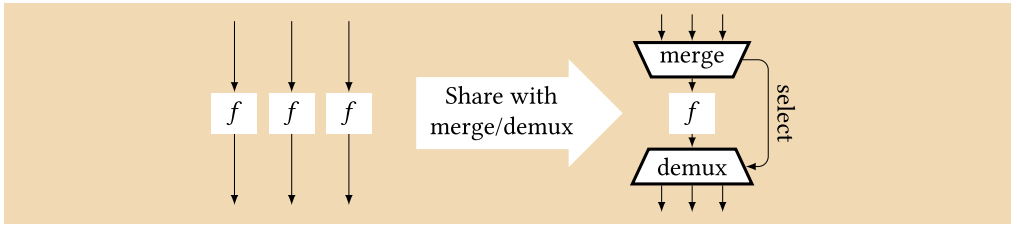
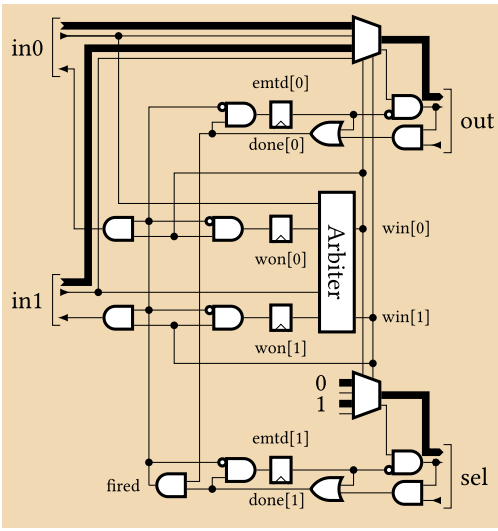


Fig. 6. A merge used to share a unit-rate subnetwork.




```
sel out = mergectrl Bool Int < in0 in1;
```

```
logic [1:0] won, win; // 2 input arb. bits
assign win = | won ? won : // Decided
             in0[0] ? 2'd1 : // in0 wins
             in1[0] ? 2'd2 : // in1 wins
                      2'd0; // No winner
initial won = 2'd0; // No winner initially
always_ff @(posedge clk)
    won <= fired ? 2'd0 : win;
logic [1:0] emtd, done; // One per output
assign done = emtd |
        ({ sel[0], out[0]} & { sel_r, out_r });
initial emtd = 2'd0; // Nothing yet emitted
always_ff @(posedge clk)
    emtd <= fired ? 2'd0 : done;
assign fired = & done;
assign {in1_r, in0_r} = fired ? win : 2'd0;
assign out = win[0] && !emtd[0] ? in0 :
             win[1] && !emtd[0] ? in1 :
                  { {W{1'bx}}, 1'd0 } ;
assign sel = win[0] && !emtd[1] ? 2'b01 :
             win[1] && !emtd[1] ? 2'b11 :
                                  2'bx0 ;
```

Fig. 7. A two-input nondeterministic merge that reports arbitration decisions on 1-bit output *sel*.

Figure 7 shows a two-input merge actor, which has two possible behaviors. If only one data input (*in0* or *in1*) provides a token, then it is routed onto the *out* port. If both data inputs provide tokens in a given cycle, then only one is routed onto *out* by the arbiter, whose implementation can vary (thus our modeling of the merge as nondeterministic). In both cases, the merge reports which input was selected via the *sel* port.

As we described in Section 2.4, Broy [4] models nondeterministic merge as a mux driven by a nondeterministic input that controls how the input streams are interleaved; our merge actor emits this selection sequence.

The circuit in Figure 7 is complex, because it generates tokens on two output channels when it fires and needs to cope with only one channel being ready. The naïve approach of insisting both outputs be ready for the actor to fire leads to circuits with combinational cycles; our circuit avoids this by allowing firing across multiple cycles and thus needs state. The fork described in Section 4.2 is similar.

An n-input merge has $n + 2$ state bits: n one-hot "*won*" bits that indicate which input won the arbitration and two *emtd* bits that indicate the *out* and *sel* outputs have already emitted tokens in this firing and should not emit any more. All of these bits reset to 0 between firings.

Our merge actor is built around an arbiter that, with a simple priority-encoding scheme, selects a valid input and declares it the winner through the one-hot *win* vector. This vector controls the multiplexers that route the winning input to *out* and its identity to *sel*. While our generated circuits are technically deterministic at the cycle level, changing the buffering on the channels may affect the behavior of the merge actor, because it responds to the cycle-level timing behavior of input sequences. As such, it is effectively nondeterministic at the level of the Kahn network.

If both *out* and *sel* are ready, then both *done* signals become true, *fired* becomes true, the winning input's *ready* is asserted, all the *won* and *emtd* registers stay at 0, and the arbiter can handle another token in the next cycle.

When an output is not ready, it sets the *emtd* bit for the other output, suppressing that output's *valid* signal in the next cycle. Furthermore, because *fired* is not asserted, the *win* vector will be loaded into the *won* register. In the next cycle, since the *won* register is non-zero, the arbiter will maintain the identity of the winner and ignore any new input tokens.

In cycles after the initial arbitration, the *win* vector holds its value and maintains a valid token on the output that has not yet been consumed. When both outputs have finally been consumed (i.e., when each is either emitted or ready), *fired* will be asserted, the winning input token is finally consumed, and the merge actor's state resets to fire again in the next cycle.

While having a nondeterministic merge that reports which input token won the arbitration is useful for resource sharing (e.g., as in Figure 6) the select output is not always needed. Our compiler (see Section 6) also provides a merge process with no additional select output, whose implementation is much simpler than that in Figure 7.

4 CHANNELS IN HARDWARE

In our specifications, a channel is an abstract mechanism that conveys a sequence of tokens generated by a process or supplied by the environment to one or more processes. Our technique allows such channels to be implemented in a variety of ways, providing various speed/area tradeoffs.

A point-to-point channel can be implemented with a direct connection, as shown in Figure 2. Such a link is the fastest and consumes the fewest resources but may produce a long combinational path that limits clock frequency. It also couples the two process' firings.

Adding a buffer to a point-to-point link decouples the firing of the upstream and downstream actors. Such buffering is mandatory on loops in the network and on channels with initial tokens (see Section 2.2). Buffering can also improve performance by breaking long combinational paths in the generated circuit, effectively pipelining them to improve throughput and clock frequency.

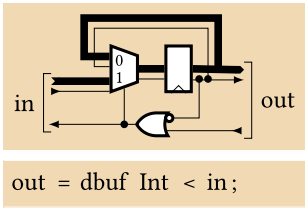
We provide *fork* blocks for implementing channels with fanout. The datapath of a fork is trivial (simply wires that fan out); the flow-control logic (i.e., for *valid* and *ready*) turns out to be fairly complicated to avoid a combinational path from *ready* to *valid*.

Choosing an optimal channel implementation is outside the scope of this article. However, we can correctly implement any channel in our specification as a single-source, feed-forward network comprised of forks, buffers, and point-to-point connections.

Below, we describe how we implement buffers and forks.

4.1 Data and Control Buffers

We provide two buffer types: a data buffer breaks a combinational path (or cycle) on the data/valid network and a control buffer does so on the ready network. Each type of buffer can hold a single data token, but their implementations differ.

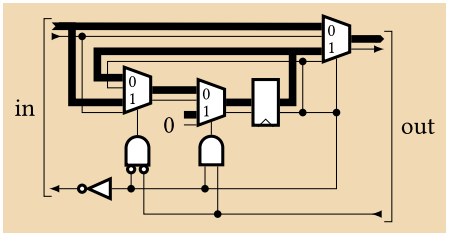



```
out = dbuf Int < in;
```

```
initial out = { {W{1'bx}}, 1'b0}; // Start empty
assign in_r = out_r || !out[0]; // Will we have space?

always_ff @(posedge clk)
  if (in_r) out <= in; // If so, save the token
```

Fig. 8. A data (pipeline) buffer, after Cao et al. [6]. This breaks combinational paths in the data/valid network.




```
out = rbuf Int < in;
```

```
initial buffer = {{W{1'bx}}, 1b'0}; // Empty
always_ff @(posedge clk)
  if (out_r && buffer[0]) // Will send?
    buffer <= { {W{1'bx}}, 1'd0}; // then clear
  else if (!out_r && !buffer[0]) // Must hold?
    buffer <= in; // then save
assign out = buffer[0] ? buffer : in;
assign in_r = !buffer[0];
```

Fig. 9. A control buffer, after Cao et al. [6]. This breaks combinational paths in the (upstream) *ready* network.

A data buffer (Figure 8) is a traditional pipeline register: it breaks the combinational path on data/valid signals, stores a single data token, and adds a clock cycle of latency. The downstream *ready* signal acts like a latch enable when the buffer holds a valid token; an upstream token is always latched when the buffer is empty. Note that a data buffer's *ready* path is combinational.

The control buffer in Figure 9 performs the more challenging task of breaking the combinational path on the *ready* network. By design, the upstream *ready* signal (*in_r*) depends only on a flip-flop output—the *valid* bit of a "spill buffer." Complementary to a data buffer, a control buffer induces a cycle of latency on the ready network, but not necessarily any on the data/valid network.

The control buffer intercepts and stores any valid token that the downstream cannot accept. The buffer in Figure 9 starts empty: Its *valid* bit is false, any valid token flows directly from *in* to *out*, and *in_r* is asserted. If *out_r* remains true, then the buffer remains empty (holds its previous contents); however, if *out_r* goes false, then the downstream will not consume any valid token, so instead any valid token on *in* is "spilled" into the buffer.

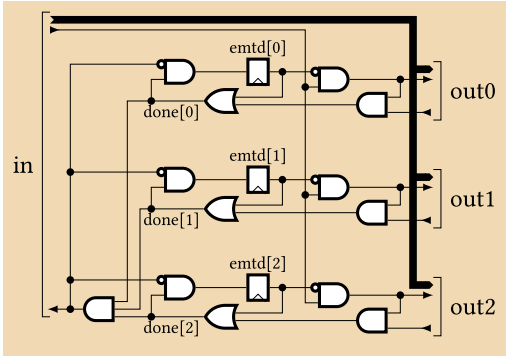
When the buffer holds a token, no token is accepted from upstream, because *in_r* is false, the buffered token is proffered downstream, and *out_r* controls whether the token will continue to be held or advanced in the next cycle.

Connecting control and data buffers back-to-back in either order breaks any combinational path that would pass through them, allowing them to be chained arbitrarily without reducing peak clock frequency. Back-to-back, these buffers behave like the latency-insensitive relay stations of Li et al. [31].

4.2 Forks

To implement channels with fanout, we use "fork" circuits that handle the flow-control logic (i.e., *valid* and *ready* signals) without introducing combinational cycles when blocks are connected.

The obvious way to implement fork—a block that waits for all its downstream actors to be ready before firing—would introduce combinational cycles when composed, because such a policy requires a combinational path from *ready* to *valid*. A block that considered *ready* inputs to control whether its outputs were valid would not be compositional.




```
out0 out1 out2 = fork Int < in;
```

```
logic [2:0] emtd, done; // Per output
initial emtd = 3'd0;    // Start clear
assign out0 = {in[W:1], in[0]&& !emtd[0]};
assign out1 = {in[W:1], in[0]&& !emtd[1]};
assign out2 = {in[W:1], in[0]&& !emtd[2]};
assign done =
  emtd | ({out2[0], out1[0], out0[0]} &
          {out2_r, out1_r, out0_r});
assign in_r = & done;

always_ff @(posedge clk)
  emtd <= in_r ? 3'd0 : done;
```

Fig. 10. A three-way fork. An output port's *emtd* flip-flop is set when the input token has been consumed on that port. All are reset after a token is consumed on every port.

Our implementation of fork avoids combinational cycles by introducing a limited amount of state. By using one flip-flop per output, our fork block allows a valid token to pass through a fork and be consumed downstream *before* it is consumed upstream. The amount of state in a *fork* block depends only on how much it fans out and not on the width of the data tokens (e.g., unlike control and data buffers).

Figure 10 illustrates our solution, which uses one flip-flop per output in a vector called *emitted*. Each *emitted* bit indicates whether the downstream consumer previously consumed the current token. If an output's *emitted* bit is set, then the fork suppresses that output's *valid* signal to avoid sending a second copy of the token to the consumer.

Initially all *emitted* bits are zero. If there is no input token, then the state is unchanged. If an input token arrives, then it is proffered on all downstream ports. If all consumers are ready, then *done* is all ones, the upstream *ready* is asserted, and the *emitted* flip-flops remain cleared.

If any consumer is not ready, then the input token is not consumed (the upstream ready is not asserted) and the ready consumers' *emitted* bits are set to one to prevent any further tokens being proffered on the outputs before the current token is consumed.

When some emitted bits are set, the upstream ready is not asserted, the input token is held, and an output token is proffered on output channels whose *emitted* bits are zero. Each output's *done* bit is asserted if the proffered token was consumed in this or a previous cycle. Once all *done* bits are true, the *emitted* bits are reset, the upstream token is consumed, and the process repeats.

5 THE ARGUMENT FOR CORRECTNESS

In this section, we argue that our circuits faithfully implement the specifications in Section 2 in that anything the hardware implementation can do is permitted by the specification. However, the reverse is not true: the hardware may deadlock because of (finite) buffer overflow where the specification would proceed. Specifically, the sequence of tokens that can be observed passing through any channel in a hardware implementation is a prefix of (but often equal to) the sequence of tokens that the Kahn fixed point semantics implies would pass through the channel.

The argument relies on our hardware behaving according to the formal notion of an actor firing. According to Equation (3), when a process finds tokens on its input sequences that match a firing rule $\mathbf{r}$, it produces tokens on its output sequences according to its firing function f, and then advances past ("consumes") the tokens identified in the firing rule by recursing on the tuple of sequences $\mathbf{t}$, which skips the tokens in the firing rule $\mathbf{r}$.

We use the *valid* signal to indicate the "next" token in sequence; a block indicates it is willing to consume a token when it asserts the *ready* signal on the port. An upstream block must continue to provide the same valid token until the downstream block is ready.

We argue that each block maintains the following inductive invariant between clock cycles: on each port, if *valid* is true, then the data wires carry the token value that appears "next" in the sequence given by the underlying Kahn network behavior; the "previous" token value was consumed during the last cycle in which both *valid* and *ready* were true (or no such token existed because the circuit was reset). Furthermore, once *valid* has been asserted, it must stay asserted until the next cycle in which *ready* is asserted. Thus, *valid* indicates the correct next token value is present; when it is accompanied by *ready* the token has been consumed. A corollary is that observing the data values on a port in cycles where both *valid* and *ready* were true gives a prefix of the sequence on that port. For the sequence of data values $\ldots, c_{t-1}, c_t, c_{t+1}, \ldots$, we might observe clock cycles with the following data (here, "X" represents garbage data):

data:	$\cdots$	c_{t-1}	c_{t-1}	c_{t-1}	X	c_t	c_t	c_{t+1}	$\cdots$
valid:	$\cdots$	1	1	1	0	1	1	1	$\cdots$
ready:	$\cdots$	0	0	1	X	0	1	0	$\cdots$

The highlighted columns are token-transfer cycles. The environmental inputs must also follow this protocol and hold valid data until it is ready to be consumed.

We also assume that when the circuit is reset, any and all initial tokens on the channels required by the specification are residing in the appropriate data or control buffers.

The **unit-rate actor** (Figure 3) preserves the invariant. If all its inputs are valid, then each carries the value of the next token on their respective sequences, the function block computes the next token in sequence on the output, which is made valid. If, additionally, the output is ready, then the inputs are also made ready, indicating the input tokens have been consumed.

For the **multiplexer** (Figure 4), if the *select* input is valid, it must carry the next token in sequence. The value of the *select* token routes the data/valid signals from the appropriate input port to the *muxed* signal internally. If *muxed* is valid, then the output carries the proper value (next token in sequence) and is set valid. If, additionally, the output is ready, then both the *select* input and the selected input are made ready and no others.

For the **demultiplexer** (Figure 5), only if both the input and select inputs have a valid token is the decoder activated and the appropriate output made valid. If, furthermore, that output is also ready, then only then are both the input and select inputs marked ready.

The **nondeterministic merge** block (Figure 7) must ensure that once it decides what the next tokens on its output should be, these values persist. When the *emitted* and *won* registers are all zero, the arbiter decides which one, if any, of the valid inputs wins the arbitration. This causes correct, valid tokens to appear on both the output and select ports. If both ports are ready, then *fired* is asserted, the winning input is also marked ready, and the *emitted* and *won* registers stay zero. If only one output port is ready, then *fired* will remain low, the ready output port will set its *emitted* bit and the *won* register will record the arbitration winner. In future cycles the token, if any, from the winning input port will continue to be routed to the output port and the corresponding value will be sent on select, but the *emitted* register will suppress the *valid* signal on the already-ready port. Note that the environment must sustain the valid token on the winning input port. If *ready* is asserted on the non-emitted port, then *fired* will be asserted, the winning input will be made ready, the registers cleared, and the process repeats.

Only buffers can hold tokens. Consider when the **data buffer** (Figure 8) is empty. The output is invalid and the *ready* output is asserted. If a valid token is proffered, then the token will be stored in the buffer at the end of the cycle, consistent with the invariant. When the data buffer is full,

valid is asserted. If the downstream *ready* is false, then the upstream ready is false and the register will hold the token. When the downstream ready is true, the upstream ready will be asserted and the token in the buffer will be overwritten. If a valid token was proffered, then it will be stored in the buffer.

If the **control buffer** (Figure 9) is empty, then the input token/valid signal is simply copied to the output and the upstream *ready* is asserted. If the downstream does not assert *ready*, then the valid upstream token, if any, will be stored in the buffer for the next cycle. If the buffer is full, then the upstream *ready* is false and the valid token is proffered on the output. If the downstream *ready* is true, then the buffer will be emptied in the next cycle.

The **fork** block (Figure 10) relies on the upstream block sustaining a valid token until it is consumed. When the *emitted* register is zero, a valid token on the input becomes a valid token on each of the outputs. Any output that is also ready asserts its respective *done* signal. If all the *done* signals are set, then the upstream *ready* is asserted and the *emitted* registers are all reset. Otherwise, each ready output sets its *emitted* bit in the next cycle. These bits suppress the *valid* signal on each of the outputs that had already asserted *ready* with the current input token. Each *done* bit becomes true if its *emitted* bit is true or if a valid token has been consumed by a ready on the output. When all the *done* bits are true, the current token is consumed from the input and the *emitted* bits are all reset.

6 DF: A TYPED LOW-LEVEL DATAFLOW LANGUAGE

To generate dataflow circuits with our approach, we developed a typed dataflow assembly language and compiler that generates synthesizable SystemVerilog. Similar to a software assembly language, our language, which we call DF, admits a one-to-one translation scheme: Each line instantiates one dataflow actor by generating the SystemVerilog code presented earlier (modifying the number of inputs, outputs, and setting their bitwidths appropriately). A DF program describes a dataflow network, which our compiler type-checks before generating the corresponding SystemVerilog following our procedure (Section 3).

Compared to coding dataflow networks directly in SystemVerilog, our DF language and compiler provides the usual advantages of a higher-level language: it makes good designs easier to express and prohibits many bad designs. It is much more succinct than the equivalent SystemVerilog would be, making it easier to write and read. It provides higher-level types: signed and unsigned Binary vectors plus algebraic data types (tagged unions) provide both abstraction to more succinctly express ideas and opportunities to catch design errors early. The network is checked for types and structure: each actor specifies constraints on the types of its input and output ports, each channel is required to have exactly one writer and one reader, and the types of the ports on a channel must match.

We use DF as a textual intermediate representation in our Haskell-to-hardware compiler [43] to both simplify debugging and give a more modular compilation flow than a pure binary representation would. Although we would have preferred to express our actors as a SystemVerilog library, SystemVerilog's polymorphism does not permit modules with a varying number of inputs or outputs (e.g., merge and fork). Furthermore, although the 2012 SystemVerilog standard includes tagged unions for implementing algebraic data types, none of the SystemVerilog tools we wanted to use (e.g., Quartus and Verilator) supported them. We may revisit this choice in the future.

Figure 11 shows a complete DF program that expresses a fragment of the GCD example from Figure 1. It starts with channel type definitions (DF has no built-in types), then lists type definitions for the various actors (e.g., for fork and op_eq), then instantiates the actors.

Figure 12 shows the syntax for DF. A program consists of channel type definitions, actor type definitions, and actor instances. We describe each below.

```
// Type definitions
data Int signed 32 ;
data Bool = False | True;
// Actor type definitions
source  a     :       > a                    ;
sink    a     : a     >                      ;
fork    a     : a     > a+                   ;
op_eq   a     : a a   > Bool                 ;
demux   a b   : a b   > b∧(variants a)       ;
mux     a b   : a b∧(variants a)   > b       ;
initbuf a (b : a) : a               > a      ;
rbuf    a     : a     > a                    ;
```

```
// Actor instances
aIn          = source  Int            <;
bIn          = source  Int            <;
aFB          = source  Int            <;
bFB          = source  Int            <;
e1b          = rbuf  Bool             < e1;
c            =  initbuf  Bool True    < e1b;
c1 c2        = fork  Bool             < c;
a            = mux Bool Int           < c1 aIn aFB;
a1 a2        = fork  Int              < a;
b            = mux Bool Int           < c2 bIn bFB;
b1 b2        = fork  Int              < b;
e            = op_eq Int              < a1 b1;
e1 e2 e3     = fork  Bool             < e;
result  aF   = demux Bool Int         < e2 a2;
_       bF   = demux Bool Int         < e3 b2;
             = sink  Int              <  result ;
             = sink  Int              < aF;
             = sink  Int              < bF;
```

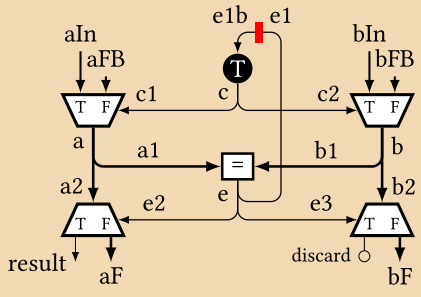


Fig. 11. A DF program describing the topology and channel types for a portion of GCD from Figure 1.

program	::=	[*typedef* | *actordef* | *instance*]*	
typedef	::=	**data** *type-id* [**signed** | **unsigned**] *int-lit* **;**	Primitive sized integer type
	|	**data** *type-id* **=** *tag-id type-id** [**|** *tag-id type-id**]* **;**	Algebraic type definition
actordef	::=	*actor-id* [*var-id* | **(** *var-id* **:** *type* **)**]* **:** *type** **>** *type** **;**	Actor type definition
instance	::=	*channel-id** **=** *actor-id* [*type-id* | *int-lit*]* **<** *channel-id** **;**	Actor instance
type	::=	*type-id*	Named type
	|	*tag-id*	Tag name (variant)
	|	*var-id*	Type variable/function name
	|	*type* **+**	One or more
	|	*type* ∧ *type*	Prescribed number of
	|	*type type*	Type function application
	|	**(** *type* **)**	Grouping

Type (*type-id*) and tag (*tag-id*) names start with an uppercase letter.
Actor (*actor-id*), channel (*channel-id*), and type variable (*var-id*) names start with a lowercase letter or an underscore (_).
Integer literals (*int-lit*) may be negative.

Fig. 12. Syntax of DF. Brackets [], bar |, and asterisk * are meta-symbols denoting grouping, choice, and zero-or-more. Bold characters, including parentheses **()**, bar **|**, and caret ∧, are tokens.

6.1 Channel Type Definitions

Each channel in a DF specification has a type indicating the type of tokens it conveys. A channel type must be defined with the *data* keyword before it can be used in a DF program, and its name must begin with an uppercase letter. Our compiler implements all types as fixed-width bit vectors in SystemVerilog.

DF's channel types are either primitive integers or composite algebraic types. An integer type definition names either a binary (**unsigned**) or two's complement (**signed**) fixed-width bit vector:

```
data Int  signed 32;        // 32-bit signed (two's complement) integer
data Char unsigned 8;       // 8-bit unsigned integer
data Uint14 unsigned 14;    // 14-bit unsigned integer
```

DF's algebraic types are similar to those in functional languages such as ML and Haskell. An algebraic type consists of one or more variants; each variant has a name ("tag") starting with an uppercase letter and a payload of zero or more data fields of specific types:

```
data Bool = False | True;       // The usual Boolean type
data Pair = Pair Int Int;       // A pair of integers
```

Whereas an integer token carries a numeric value, a token of an algebraic type carries one variant and its associated payload data, which may be other algebraic types and integers. Our compiler encodes algebraic types as a tagged union: a tag field indicates the variant followed by enough bits to hold the largest payload.

Algebraic types are flexible enough to subsume other languages' enumerated types, tuples/structures, and objects with inheritance. Although hierarchy is allowed, recursive types are prohibited. As in software, recursive types may be expressed with pointers encoded as integers:

```
data Tree = Branch Uint14 Uint14  // Binary tree with 14-bit pointers.
          | Intleaf  Int          // Leaves may be 32-bit integers
          | Boolleaf Bool         // or Booleans
```

6.2 Actor Instances and Type Definitions

An actor instance adds a new actor to the network and consists of a list of output channels, an actor name, a list of zero or more arguments to be passed to that actor's type definition, and a list of input channels:

output-channel ... = *actor-name type/literal-argument* ... < *input-channel* ... ;

Unlike other network specification languages such as Verilog, DF does not need to name each actor instance; the names of an instance's channels uniquely identify that instance, since connecting two actors to exactly the same inputs and outputs is nonsensical (and illegal).

An actor type definition specifies the type and number of ports for an actor by providing a unique actor name, a list of parameters, and lists of input and output port types:

actor-name parameter ... : *input-port-type* ... > *output-port-type* ... ;

Our actors support parametric polymorphism by taking zero or more named parameters, which are typically used to specify the type and number of ports for that actor. A parameter with just a name (e.g., "a") is a type variable that ranges over channel types. A parameter may also be constrained to constants of a particular channel type (e.g., "a : Int") or variant tags of an algebraic type (e.g., "a : tag Bool"). Actor instances provide concrete type and constant arguments to resolve any parametricity in a definition.

We use a regular-expression-like syntax inspired by Hosoya et al.'s type system [23, 24] to specify the number and type of an actor's input and output ports. A type name (e.g., "Bool") denotes a single port of that type, while a type variable (e.g., "a") represents a single port of polymorphic type. The ∧ and + operators let us assign types to multiple ports at once: an expression of the form $t \wedge n$ means n ports of type t (where n is an integer expression), and the postfix + operator denotes one or more ports of a given type (e.g., "Bool+"). To avoid ambiguity, the + operator may only be used once in the input channels and once in the output channels.

Below are some actor type definition examples. A *source* is an input connection from the environment that supplies tokens of a given type. It adds ports to the SystemVerilog module generated for the network. An *op_eq* is a two-input polymorphic comparator (Section 3.2) that emits True when its inputs are equal and False otherwise. An *op_add* takes two objects of the same type, sums their bits, and returns an object of the same type. A *buf* is a data/control buffer pair that can buffer any type of token (Section 4.1). An *initbuf* is a *buf* that starts with an initial token whose value is the second parameter. A *fork* is a polymorphic single-input actor that can have one or more outputs (Section 4.2).

```
source   a          :      > a;     // Actor with a single output of any type
op_eq    a          : a a  > Bool;  // Two inputs of the same type and a Bool output
op_add   a          : a a  > a;     // Two inputs and an output, all the same type
buf      a          : a    > a;     // Single input and single output ports are of the same type
initbuf  a (b : a)  : a    > a;     // Second parameter is a constant of type a
fork     a          : a    > a+;    // One input; one or more outputs, all the same type
```

We also provide three built-in type functions that actor definitions may use to help define channel types. The *variants* function returns the number of variants of a channel type. This is used with the ∧ operator to specify a number of ports, e.g., for a multiplexer, which uses the variant sent to it on a *select* input to choose an input. Below, because the multiplexer's *select* input takes the three-variant type *Tri*, the mux has three inputs.

```
data Int signed 32;
data Tri = One | Two | Three;
mux a b : a b∧(variants a) > b;
output = mux Tri Int < select input1 input2 input3;
```

The *tag* and *variant_fields* functions work in tandem to let us define a *variant* actor that assembles the payload fields of an algebraic type object to produce an instance of that type. Specifically, the *tag* function specifies that a parameter should be a variant of a given type, and passing that parameter to the *variant_fields* function yields a list of channel types corresponding to that variant's fields. For example, the following fragment constructs both variants of an *OptPair* type:

```
data Int signed 32;
data OptPair = Pair Int Int | Null;
source a : > a;
variant a (b : tag a) : ( variant_fields b) > a;
i1 = source Int < ;
i2 = source Int < ;
p = variant OptPair Pair < i1 i2; // Pair is a variant of OptPair, payload of two Ints
n = variant OptPair Null < ;      // Null is a variant of OptPair, no payload
```

The *destruct* actor works in reverse, extracting the payload from the given variant:

```
destruct a (b: tag a) : a > ( variant_fields b);
o1 o2 = destruct OptPair Pair < p; // o1 and o2 will be Ints
```

6.3 Checking DF Specifications

The DF compiler has two main tasks: to verify a DF program is correctly typed and to translate it into SystemVerilog. We described how the translation is performed in Section 3 and Section 4; here we discuss the rules our compiler uses to check types.

The first set of checks concerns name usage. A DF program has four global namespaces: type names, tag names, actor names, and channel names. Type and tag names must start with an uppercase letter, while actor and channel names start with a lowercase letter or an underscore. Each type, tag, and actor defined must have a globally unique name within its respective namespace (e.g., we can define a type and a tag of the same name).

To enforce the point-to-point nature of communication channels, the DF compiler requires each named channel to appear exactly twice in a network: once as an output and once as an input. It would be possible to relax this requirement and allow a channel to be connected as an input to multiple actors. The compiler would implement such channels with *fork* circuitry (Section 4.2).

Once all names have been checked, the compiler first validates each data type definition individually by checking that each variant's payload only carries defined types, followed by an overall check that no type is recursively defined.

To validate an actor type definition, the compiler ensures that both its parameters and channel types follow various rules. Each parameter name must be unique, but only for the actor being defined. Constraints on parameter types can only refer to earlier parameters for that actor (e.g., "a : b" is only valid if "b" was defined earlier in the parameter list), and such constraints must resolve to either a channel type or the tags of a type.

The channel types for an actor type definition must be consistent. Each must resolve to either a named type, a type variable, or the + or ∧ operators applied to one of these. At most one + operator may appear among the inputs and at most one may appear in the outputs. The right argument of each ∧ operator must resolve to an integer. The *variants* and *tag* functions may only be applied to a type, while the *variant_fields* may only be applied to a tag.

The compiler checks each actor instance in two steps: first, actual arguments are bound to each of the actor's parameters, and then types are assigned to each channel (both inputs and outputs). Binding arguments to parameters is done in the usual way: the nth argument is bound to the nth parameter provided its type is consistent, i.e., a parameter that is a type variable only accepts a concrete type (name), a parameter constrained to a channel type must be passed a literal of that type, and a parameter constrained to be a tag of a particular channel type must be given such a tag.

If binding arguments to parameters succeeds, then the second step is a matching process that assigns types to input and output channels. Parameters are used to resolve the type (and number, in the case of the ∧ operator) of expressions without a + operator, but the presence of an expression with a + operator complicates things. With a + operator, our procedure assigns the channels before the + to the channels at the beginning of the input or output list, the channels after the + to those at the end of the list, and the remainder to the range denoted by the +. Multiple + operators in a list of channels would introduce ambiguity, so we prohibit them. Multiple ∧ operators, however, are allowed, because they prescribe a specific number of channels when resolved.

Finally, we verify that the type assigned to each channel when it appears as an output is the same as the type assigned to the channel when it appears as an input.

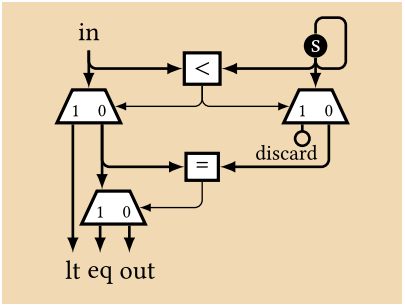


Fig. 13. The splitter component of a Conveyor. This network partitions tokens arriving on the input stream *in* by comparing each input value against a "split" value: the initial token *s*. Each input token is sent out on one of three ports depending on whether it is less than (*lt*), equal to (*eq*), or greater than (*out*) the split value.

7 EVALUATION

A designer can use our blocks to implement a dataflow network and adjust buffering to affect area and performance without changing functionality, although insufficient buffering may introduce deadlock. To verify this, we created dataflow networks (both with DF and with the compiler of Townsend et al. [43]) buffered them both randomly and manually, simulated the resulting circuits to check that each functioned correctly, and calculated the circuits' highest clock rate when synthesized on an FPGA.

We both simulated the function of each circuit with Verilator 3.874 to verify that our circuits operate correctly and are free of combinational cycles and synthesized each circuit using Altera's Quartus 15.0, targeting a modest-performance Cyclone V 5CGXFC7C7F23C8 FPGA with 56480 ALMs, to estimate the maximum operating frequency and resource usage of the design.

7.1 Experimental Networks

We experimented with the five applications described below. We manually coded the first three networks in our dataflow language; the last two networks were synthesized from small Haskell programs using the translation of Townsend et al. [43].

GCD. This is Figure 1's network made to compute gcd(100,2) with 32-bit integers.

Conveyor. This network performs range partitioning [45]. The design chains n splitters (Figure 13) to partition an input stream into $2n + 1$ output streams (e.g., 10 splitters yield a 21-way Conveyor design). Each splitter routes tokens to its outputs depending on how each token compares to the splitter's value. We feed the Conveyor an input sequence of 32-bit numbers $(1, \ldots, 10000)$ and set the ith splitter value to $10000/(i + 1)$. To limit I/O pins, we merge the Conveyor's outputs with a chain of merge actors to produce a single (nonsensical) 32-bit output stream.

Bitonic Sorting Network (BSN). This sorts a fixed number of values with two-input comparators that operate in parallel. Figure 14 shows the dataflow network for a comparator and an eight-input BSN. Each comparator takes in a pair of tokens and routes the smaller to its lower output and the larger to its upper, either by passing the tokens straight through or swapping them. We execute this network on 10 sets of eight 8-bit values and merge the sorted numbers with an 8-input adder (again, to limit I/O pins).

MergeSort and TreeSort. These are recursive sorting algorithms that use memories to store their data structures (lists and trees) and the "continuation" objects that implement their recursion.

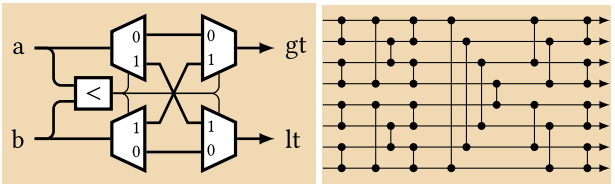


Fig. 14. Bitonic sorting: the network on the left routes the larger of two input tokens to the top output and the smaller to the bottom. These are the vertical lines in the eight-element bitonic sorting network on the right (after Cormen et al. [10]).

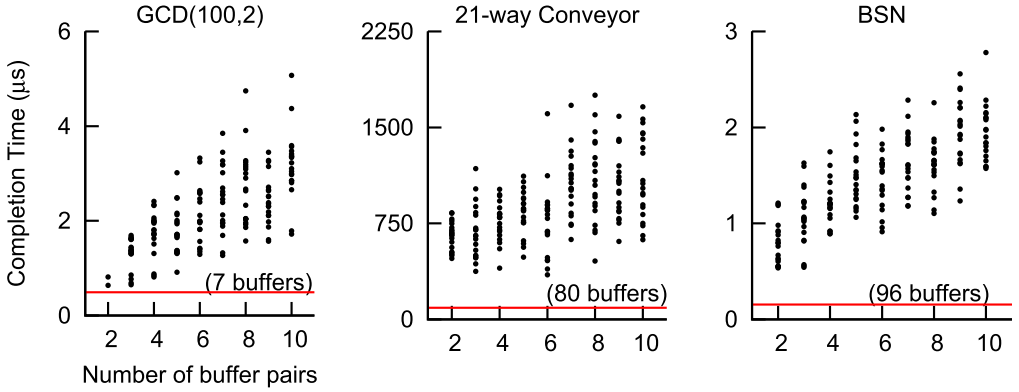


Fig. 15. Completion times under random buffer placement. Horizontal lines labeled with a buffer count indicate the completion time of the best manual design.

We feed each network a list of 20 32-bit integers. We limited the input size, because the circuits generate a large number of intermediate structures and our memories are not currently garbage collected.

7.2 Random Buffer Allocation

We first employ random buffer allocation, not because it produces efficient designs but to show that buffers may be added arbitrarily without affecting functionality thereby facilitating design space exploration. Given unbuffered GCD, BSN, and 21-way Conveyor networks, we assign data buffers to store the initial tokens from their specifications (GCD and Conveyor) and place a control buffer on the same channels to break a combinational cycle.

We next assign between two and ten control/data buffer pairs on randomly chosen channels, discarding any implementation that produces premature deadlock or leaves a combinational cycle. All remaining implementations compute the same result. Figure 15 shows the completion time of each of these implementations in microseconds: cycles divided by maximum frequency (MHz).

7.3 Manual Buffer Allocation

In Figure 15, we also plot the completion time for the best design we could devise manually (the horizontal lines). Naturally, these are much better than what random buffering produced.

Figure 16 depicts our best manual buffer placements. Each black bar represents a data buffer; red represents a control buffer. For space, we only depict 2 representative splitters (of 10) in the Conveyor.

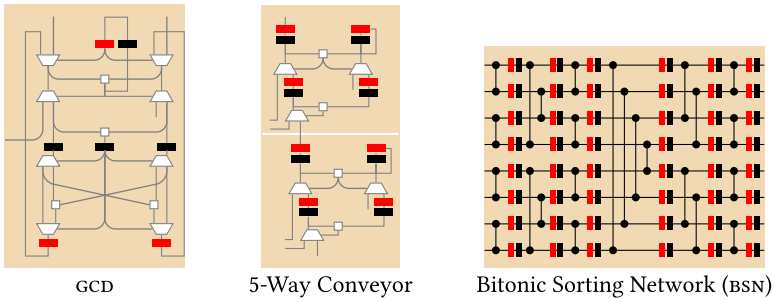


Fig. 16. Buffering our networks. Red bars represent control buffers; black for data. Each cycle requires both buffers.

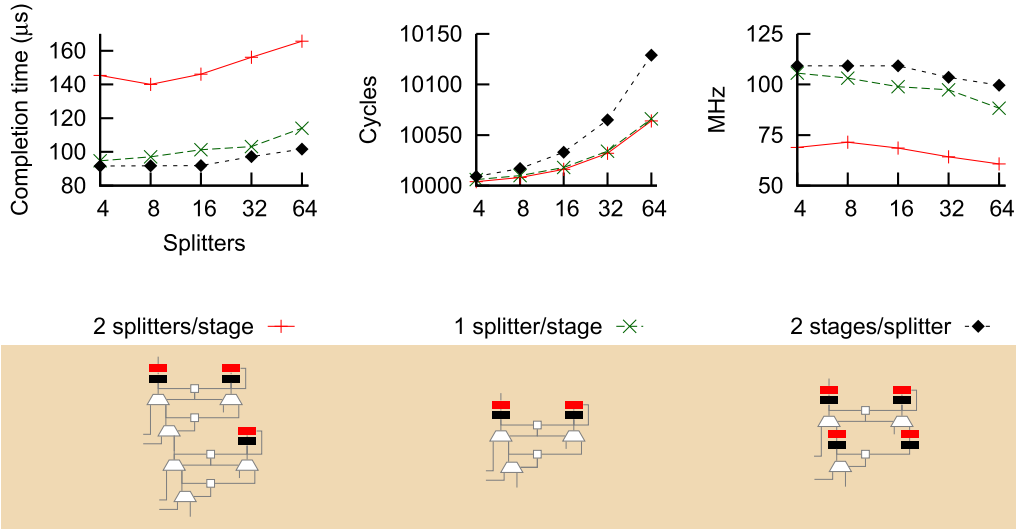


Fig. 17. Three Conveyor pipelining strategies: Doubling the number of stages has only a small effect on total completion time.

We manually implemented 6-stage and 20-stage BSN and Conveyor networks, respectively. Each stage adds only a single additional cycle to the total execution (since no stalls occur) while substantially increasing the clock frequency (103MHz for BSN; 109MHz for Conveyor) to reduce completion time.

We found separating data and control buffers improved performance for the GCD example. We initially placed the two buffer types together at the bottom of the network, but found splitting and moving them as shown in Figure 16 improved the frequency from 79 MHz to 109 MHz.

7.4 Pipelining the Conveyor

Over Conveyors with 4 to 64 splitters, we experimented with the three pipelining strategies shown in Figure 17: two splitters per stage, one splitter per stage, and two stages per splitter.

The graphs show that the two stages/splitter design provides the best overall performance on our workload, followed closely by the one stage/splitter. For one stage/two splitters, the barely noticeable reduction in the number of cycles required is swamped by the 40% reduction in maximum clock frequency.

Table 1. Comparing Manually Coded RTL SystemVerilog with that Generated from DF

	Frequency (MHz)			Area (ALMs)			Registers		
Application	**Manual**	**DF**	**Ratio**	**Manual**	**DF**	**Ratio**	**Manual**	**DF**	**Ratio**
GCD	139	109	0.784	107	234	2.19	67	161	2.4
Conveyor (1 stage)	223	115	0.515	68	61	0.9	165	79	0.48
Bitonic sort	194	103	0.531	327	1212	3.71	434	1944	4.45
MergeSort	137	52	0.38	206	3160	15.3	354	5564	15.7

7.5 Memory in Dataflow Networks

Our goal is to use our dataflow networks to implement realistic algorithms with irregular memory access patterns. MergeSort and TreeSort both meet these criteria: sorting is a ubiquitous problem, and these algorithms employ pointer-based data structures.

We can incorporate memories in our networks with block RAM ("BRAM") actors along with actors that maintain an address pointer (one per BRAM) and route memory requests and results to and from the rest of the network. We employ a separate BRAM per object type, allowing us to tailor its width to the size of the object.

We modify the translation of Townsend et al. [43] to insert memory actors into the MergeSort and TreeSort networks and translate them to SystemVerilog. Each BRAM actor becomes a bit vector array with 8-bit addresses, which we access with a basic memory model: given an address and an optional write enable signal with data, the array produces the data at that address before writing in new data if the write enable signal is high. We place two data/control buffers around each BRAM to impose a two-cycle latency per Altera's recommendations.

The TreeSort circuit operates at a higher frequency but uses more memory, because its (two-pointer) tree objects are wider than (one-pointer) list objects: it completes in 94.8μs, operates at 54MHz, and uses 3,330 ALMs and 8.7kB of memory, while MergeSort takes 82.9μs, running at 52MHz with 3,160 ALMs and 5.9kB. These are meant to demonstrate that our formalism readily accommodates memory and are not high-performance sorters.

7.6 Overhead of Our Method

Table 1 lists some statistics on a subset of our designs; we compare the output of our compiler to manually coded RTL SystemVerilog. The numbers we report here come from Quartus running as described earlier.

These experiments attempt to quantify the costs of using our distributed buffering strategy, but are not so simple, because they often compare different designs due to the differences between dataflow and combinational logic. The GCD example mostly reflects the cost of additional buffers. While the manual implementation can operate with just two 32-bit data buffers, the DF version requires four: two on the data network and two on the control network. The number of buffers is directly correlated to registers used, which is shown in the 2.4 ratio between manual and dataflow. This also affects the area due to the extra logic required for the handshake protocols.

Both versions of the Conveyor example use almost the same area, but this time extra buffers were placed around the inputs and outputs of the manual implementation, which is why in this case the dataflow has half as many registers as the manual implementation.

Our Bitonic sort example is much larger and slower than the manual implementation because of the insertion of more buffers than necessary and the additional flow-control logic to handle when only certain outputs are ready to accept data. The manual implementation does not support backpressure on the outputs.

The MergeSort example diverges most widely, but this is because the two implementations take a very different approach to implementing the algorithm. The manual implementation assumes the data arrives in an array and uses random access to that array; the DF implementation is synthesized from a Haskell program that recursively sorts two linked lists. The different speeds, areas, and registers of the two implementations reflect the overhead of handling lists and recursion.

8 RELATED WORK

Surprisingly little has been written about the details of synthesizing hardware from a dataflow model as dynamic as ours. Tripakis et al. [44] surveys a number of dataflow-to-hardware projects, but most have focused on statically schedulable models such as SDF that do not support actors that make data-dependent decisions such as mux and demux. For example, the LID work of Carloni et al. [8] inspired us but only considers unit-rate actors.

Perhaps closest in spirit to our work is that of Janneck et al. around CAL [15]: a rich, functional-inspired language for expressing dataflow process actors and networks. Janneck et al. [3, 26] have a synthesis system for such networks, although little has been published about its internals. CAL treats nondeterminism differently than we: CAL actors are nondeterministic in general; we consider only a nondeterministic merge actor that reports its choices. Thavot et al. [42] synthesize hybrid hardware/software systems from CAL.

The Elastic Systems of Cortadella et al. [11, 12] are also networks based on tokens and handshaking, but also include the notions of speculation and anti-tokens. Their focus has been more on processor datapaths instead of synthesis of high-level algorithms.

The valid-ready handshake protocol we use is standard, especially in asynchronous systems [39]. Intel's 8008 used a similar protocol in 1972 to wait for slow memory [25], but the protocol likely appeared even earlier. We took our inspiration from Li et al. [31], but the same protocol can be found in Cortadella et al. [11, 12], Dimitrakopoulos et al. [14], ARM's AXI4-stream protocol [1], and the FIFOs provided in Altera and Xilinx FPGAs.

Our use of a merge to share a computational resource is also standard. We were inspired by Sharp and Mycroft [40], who use a similar technique to share function invocations, but our technique further allows pipelining. Dennis [13] and Zebelein et al. [46] propose a more complicated technique in which colored tokens allow multiple simultaneous invocations of a function and allow tokens of different colors to overtake each other. In the future, we plan to explore introducing such "virtual channels," although fear the overhead of examining token colors and adding bypass paths could be excessive.

Possignolo et al. [36] also consider token/handshaking pipelines for processor design, but the actors they present are not compositional. They start with a synchronous circuit with no handshaking and transform it into a patient dataflow network by introducing four actor circuits (variants of our unit-rate, fork, demux, and merge) based on designer annotations. Although their fork and merge actors can induce deadlock in general, they provide a set of design rules that prevent deadlock. They use Colored Petri Nets to model throughput, something we may be able to adopt.

ForSyDe [32, 37, 38] has the ability to implement systems that communicate between processes using a handshake protocol like ours, but they avoid handshaking-induced combinational cycles by always inserting delays on channels, making their communication typically slower than ours. Moreover, these channels are not user-visible: their system presents the user with a synchronous model of computation (i.e., unit-rate dataflow with no decisions). This makes it more difficult for a user to specify variable-rate processes in ForSyDe.

Low overhead is a novelty of our work, which makes simple actors such as adders and multiplexers practical; other actor-based hardware environments use more heavyweight communication that is only practical when all actors are large. For example, Keinert et al.'s [28]

SystemCoDesigner employs behavioral synthesis (Forte's Cynthesizer product) to synthesize hardware for coarse-grained dataflow actors expressed in SystemC with the SysteMoC library [16]. Inter-actor communication is done through FIFOs taken from a library [22].

9 CONCLUSIONS

We presented a formal model of dataflow networks that admit data-dependent actors and nondeterministic merge, and we showed how to implement these networks in hardware. The Kahn principle allows us to insert buffers without changing function, only performance. We argued for the correctness of our implementations and validated them with random design space exploration.

ACKNOWLEDGMENTS

The National Science Foundation funded this work (CCF-1162124).

REFERENCES

[1] ARM. 2010. *AMBA 4 AXI4-Stream Protocol Specification Version 1.0.*

[2] Twan Basten and Jan Hoogerbrugge. 2001. Efficient execution of process networks. In *Communicating Process Architectures (CPA)*, Alan Chalmers, Majid Mirmehdi, and Henk Muller (Eds.). IOS Press, Bristol, UK, 1–14.

[3] Endri Bezati, Marco Mattavelli, and Jörn W. Janneck. 2013. High-level synthesis of dataflow programs for signal processing systems. In *Proceedings of the International Symposium on Image and Signal Processing and Analysis (ISPA'13)*. IEEE, 750–755. DOI: https://doi.org/10.1109/ISPA.2013.6703837

[4] Manfred Broy. 1988. Nondeterministic data flow programs: How to avoid the merge anomaly. *Sci. Comput. Program.* 10, 1 (Feb. 1988), 65–85. DOI: https://doi.org/10.1016/0167-6423(88)90016-0

[5] Joseph Tobin Buck. 1993. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model.* Ph.D. Dissertation. University of California, Berkeley.

[6] Bingyi Cao, Kenneth A. Ross, Martha A. Kim, and Stephen A. Edwards. 2015. Implementing latency-insensitive dataflow blocks. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE'15)*. IEEE, 179–187.

[7] Luca P. Carloni. 2006. The role of back-pressure in implementing latency-insensitive systems. *Electr. Not. Theor. Comput. Sci.* 146, 2 (2006), 61–80.

[8] Luca P. Carloni, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli. 2001. Theory of latency-insensitive design. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 20, 9 (Sep. 2001), 1059–1076.

[9] Josep Carmona, Jordi Cortadella, Mike Kishinevsky, and Alexander Taubin. 2009. Elastic circuits. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 28, 10 (Oct. 2009), 1437–1455. DOI: https://doi.org/10.1109/TCAD.2009.2030436

[10] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. *Introduction to Algorithms* (McGraw-Hill, New York, NY).

[11] Jordi Cortadella, Marc Galceran-Oms, and Mike Kishinevsky. 2010. Elastic systems. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE'10)*. IEEE, 149–158. DOI: https://doi.org/10.1109/MEMCOD.2010.5558639

[12] Jordi Cortadella, Mike Kishinevsky, and Bill Grundmann. 2006. SELF: Specification and design of synchronous elastic circuits. In *Proceedings of the ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*. ACM, New York, NY, 6.

[13] Jack B. Dennis. 1974. First version of a data flow procedure language. In *Programming Symposium*, Lecture Notes in Computer Science, Vol. 19. Springer, Berlin, 362–376. DOI: https://doi.org/10.1007/3-540-06859-7_145

[14] Giorgos Dimitrakopoulos, Anastasios Psarras, and Ioannis Seitanidis. 2015. *Microarchitecture of Network-on-Chip Routers: A Designer's Perspective*. Springer, Berlin.

[15] Johan Eker and Jörn W. Janneck. 2003. *CAL Language Report: Specification of the CAL Actor Language.* Technical Report UCB/ERL M03/48. EECS Department, University of California, Berkeley.

[16] Joachim Falk, Christian Haubelt, and Jürgen Teich. 2006. Efficient representation and simulation of model-based designs in SystemC. In *Proceedings of the Forum on Specification and Design Languages (FDL'06)*, Vol. 6. ECSI, Darmstadt, 129–134.

[17] G. R. Gao, R. Govindarajan, and Prakash Panangaden. 1992. Well-behaved dataflow programs for DSP computation. In *Proceedings of the International Conference on Acoustics, Speech, & Signal Processing (ICASSP'92)*, Vol. 5. IEEE, 561–564. DOI: https://doi.org/10.1109/ICASSP.1992.226558

[18] Marc Geilen and Twan Basten. 2003. Requirements on the execution of kahn process networks. In *Proceedings of the European Symposium on Programming (ESOP'03)*, Lecture Notes in Computer Science, Vol. 2618. Springer, Berlin, 319–334. DOI: https://doi.org/10.1007/3-540-36575-3_22

[19] Marc Geilen, Twan Basten, and Sander Stuijk. 2005. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Proceedings of the 42nd Design Automation Conference*. ACM, New York, NY, 819–824. DOI: https://doi.org/10.1145/1065579.1065796

[20] Marc Geilen and Sander Stuijk. 2010. Worst-case performance analysis of synchronous dataflow scenarios. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'10)*. ACM, New York, NY, 125–134.

[21] Pieter H. Hartel, Theo C. Ruys, and Marc C. W. Geilen. 2008. Scheduling optimisations for SPIN to minimise buffer requirements in synchronous data flow. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD'08)*. IEEE, Los Alamitos, CA, 161–170. DOI: https://doi.org/10.1109/FMCAD.2008.ECP.25

[22] Christian Haubelt, Joachim Falk, Joachim Keinert, Thomas Schlichter, Martin Streubühr, Andreas Deyhle, Andreas Hadert, and Jürgen Teich. 2007. A SystemC-based design methodology for digital signal processing systems. *EURASIP J. Embed. Syst.* 2007, 1 (2007), 22. DOI: https://doi.org/10.1155/2007/47580

[23] Haruo Hosoya and Benjamin Pierce. 2001. Regular expression pattern matching for XML. *ACM SIGPLAN Not.* 36, 3 (2001), 67–80.

[24] Haruo Hosoya and Benjamin C. Pierce. 2003. XDuce: A statically typed XML processing language. *ACM Trans. Internet Technol.* 3, 2 (May 2003), 117–148. DOI: https://doi.org/10.1145/767193.767195

[25] Intel Corporation. 1972. *8008 8-Bit Parallel Central Processor Unit Users Manual*. Intel Corporation, Santa Clara, CA.

[26] Jörn W. Janneck, Ian D. Miller, David B. Parlour, Ghislain Roquier, and Matthieu Wipliez Mickaël Raulet. 2009. Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study. *J. Sign. Process. Syst.* 63, 2 (Jul. 2009), 241–249. DOI: https://doi.org/10.1007/s11265-009-0397-5

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

[28] Joachim Keinert, Martin Streubühr, Thomas Schlichter, Joachim Falk, Jens Gladigau, Christian Haubelt, Jürgen Teich, and Michael Meredith. 2009. SystemCoDesigner—An automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. *ACM Trans. Des. Autom. Electr. Syst.* 14, 1, Article 1 (Jan. 2009), 23 pages. DOI: https://doi.org/10.1145/1455229.1455230

[29] Edward A. Lee and Eleftherios Matsikoudis. 2008. The semantics of dataflow with firing. In *From Semantics to Computer Science: Essays in Memory of Gilles Kahn*. Cambridge University Press, Cambridge, UK, Chapter 4, 71–94.

[30] Edward A. Lee and Thomas M. Parks. 1995. Dataflow process networks. *Proc. IEEE* 83, 5 (May 1995), 773–801. DOI: https://doi.org/10.1109/5.381846

[31] Cheng-Hong Li, Rebecca Collins, Sampada Sonalkar, and Luca P. Carloni. 2007. 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'07)*. IEEE, 13–22.

[32] Zhonghai Lu, Ingo Sander, and Axel Jantsch. 2002. A case study of hardware and software synthesis in ForSyDe. In *Proceedings of the International Symposium on System Synthesis (ISSS'02)*. ACM, 86–91. DOI: https://doi.org/10.1145/581199.581219

[33] Orlando Moreira, Twan Basten, Marc Geilen, and Sander Stuijk. 2010. Buffer sizing for rate-optimal single-rate dataflow scheduling revisited. *IEEE Trans. Comput.* 59, 2 (2010), 188–201. DOI: https://doi.org/10.1109/TC.2009.155

[34] Thomas M. Parks. 1995. *Bounded Scheduling of Process Networks*. Ph.D. Dissertation. University of California, Berkeley.

[35] Keshav Pingali and Arvind. 1985. Efficient demand-driven evaluation. Part 1. *ACM Trans. Program. Lang. Syst.* 7, 2 (1985), 311–333. DOI: https://doi.org/10.1145/3318.3480

[36] Rafael T. Possignolo, Elanz Ebrahimi, Haven Skinner, and Jose Renau. 2016. Fluid pipelines: Elastic circuitry meets out-of-order execution. In *Proceedings of the IEEE International Conference on Computer Design (ICCD'16)*. IEEE, 233–240. DOI: https://doi.org/10.1109/ICCD.2016.7753285

[37] Ingo Sander. 2003. *System Modeling and Design Refinement in ForSyDe*. Ph.D. Dissertation. Royal Institute of Technology, Stockholm, Sweden.

[38] Ingo Sander and Axel Jantsch. 1999. System synthesis based on a formal computational model and skeletons. In *Proceedings of the IEEE Computer Society Workshop on VLSI*. IEEE, 32–39. DOI: https://doi.org/10.1109/IWV.1999.760467

[39] Charles L. Seitz. 1980. System timing. In *Introduction to VLSI Systems*, Carver Mead and Lynn Conway (Eds.). Addison-Wesley, Reading, MA, Chapter 7, 218–262.

[40] Richard W. Sharp and Alan Mycroft. 2000. *The FLaSH Compiler: Efficient Circuits from Functional Specifications*. Technical Report tr.2000.3. AT&T Laboratories Cambridge.

[41] Sander Stuijk, Marc C. W. Geilen, and Twan Basten. 2008. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Trans. Comput.* 57, 10 (2008), 1331–1345. DOI: https://doi.org/10.1109/TC.2008.58

[42] Richard Thavot, Romuald Mosqueron, Julien Dubois, and Marco Mattavelli. 2009. Hardware synthesis of complex standard interfaces using CAL dataflow descriptions. In *Proceedings of Design and Architectures for Signal and Image Processing (DASIP'09)*. ECSI, Sophia Antipolis, France, 127–134.

[43] Richard Townsend, Martha A. Kim, and Stephen A. Edwards. 2017. From functional programs to pipelined dataflow circuits. In *Proceedings of Compiler Construction (CC'17)*. ACM, New York, NY, 76–86. DOI: https://doi.org/10.1145/3033019.3033027

[44] Stavros Tripakis, Rhishikesh Limaye, Kaushik Ravindran, and Guoqiang Wang. 2014. On tokens and signals: Bridging the semantic gap between dataflow models and hardware implementations. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*. IEEE, 51–58. DOI: https://doi.org/10.1109/SAMOS.2014.6893194

[45] Lisa Wu, Orestis Polychroniou, Raymond J. Barker, Martha A. Kim, and Kenneth A. Ross. 2014. Energy analysis of hardware and software range partitioning. *ACM Trans. Comput. Syst.* 32, 3 (Aug. 2014), 8. DOI: https://doi.org/10.1145/2638550 24 pages.

[46] Christian Zebelein, Christian Haubelt, Joachim Falk, Tobias Schwarzer, and Jürgen Teich. 2014. Model-based actor multiplexing with application to complex communication protocols. In *Proceedings of Design, Automation, and Test in Europe (DATE)*. IEEE, 216–219. DOI: https://doi.org/10.7873/DATE.2014.229

Received December 2017; revised May 2018; accepted August 2018