Compositional Dataflow Circuits

Stephen A. Edwards
Columbia University
New York, New York
sedwards@cs.columbia.edu

Richard Townsend
Columbia University
New York, New York
rtownsend@cs.columbia.edu

Martha A. Kim
Columbia University
New York, New York
martha@cs.columbia.edu

ABSTRACT

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 still compute the same function without introducing spurious combinational cycles. As such, inserting or removing buffers affects the performance but not the functionality of our networks, which we demonstrate through experiments that show how design space can be explored.

CCS CONCEPTS

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

KEYWORDS

Kahn Networks, High-level synthesis, Dataflow

ACM Reference Format:
Stephen A. Edwards, Richard Townsend, and Martha A. Kim. 2017. Compositional Dataflow Circuits. In *Proceedings of MEMOCODE '17, Vienna, Austria, September 29-October 2, 2017,* 10 pages.
https://doi.org/10.1145/3127041.3127055

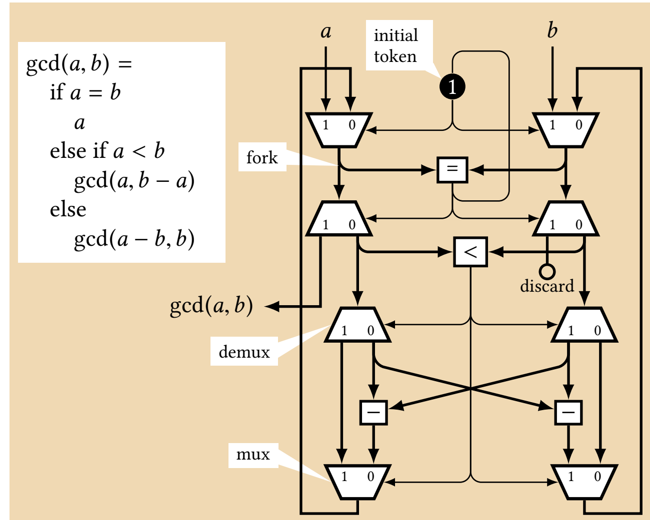


Figure 1: A recursive definition of Euclid's greatest common divisor algorithm and a dataflow network implementing it.

1 INTRODUCTION

Dataflow networks are a natural model for parallel, distributed computation [13]. Processes in a network execute in parallel and communicate via sequences of atomic tokens passed over unbounded channels. These networks are particularly suited to specifying hardware designs because of their "patience": process speed has no affect 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 [12]. 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 and loops but increases 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 long signal lines) and enables the arbitrary insertion and removal of buffering. This accommodates blocks with arbitrary latency (e.g., memories) and enables designers to adjust the number of pipeline stages, even in the presence of feedback.

Although bounded buffering will not affect the sequence of data values passed between processes (due to the Kahn model), it may introduce deadlock in our circuits. Preventing such deadlock via buffering is undecidable in general [14], but we are confident that effective heuristics can be developed. As is typical of synthesis problems, most buffer optimization problems are NP-complete (e.g., finding the minimum number of buffers necessary to break all cycles is the feedback arc set problem), but as usual, we believe there must be effective heuristics that will produce acceptable (but not always optimal) results on a wide variety of interesting networks (but not all). We defer the optimal buffer insertion problem to later work.

Fig. 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, the output will be $5 = \text{gcd}(100, 45)$ followed by $7 = \text{gcd}(56, 49)$. The 3 on b is ignored because no mating token ever arrives on a.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MEMOCODE '17, September 29-October 2, 2017, Vienna, Austria
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5093-8/17/09…$15.00
https://doi.org/10.1145/3127041.3127055

In this network, an initial "1" 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), the comparator emits a 1, causing the demultiplexers to emit the a token as the result and discard the b token. The 1 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 a 0, 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. Fig. 14 shows our preferred buffering of this GCD network.

We make the following contributions:

- circuits, for a small, rich family of data-dependent dataflow actors, that can be composed with or without buffering without introducing spurious combinational cycles;
- 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;
- a novel, "safe" way to implement nondeterministic merge for managing shared resources; and
- experiments that show how buffering may be added to explore the design space without affecting functionality.

We present our work in four pieces. First, we formalize our unbounded dataflow network specifications (§2). Our main contribution is second: how to implement bounded variants of these specifications in hardware. We describe implementing processes in §3 and channels in §4. Third, in §5, we argue why our circuits faithfully implement our specifications. Fourth, we present experimental results that demonstrate that our circuits work and facilitate design-space exploration in §6.

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 [12] provides the framework, Lee and Matsikoudis [13] show how to model firing rules, and the model of nondeterministic merge is due to Broy [2].

2.1 Kahn Networks

A Kahn network passes around atomic *tokens* drawn from a set Σ. 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. The empty sequence ϵ is included in this set, since it contains zero tokens and thus is finite ($\epsilon \in \Sigma^*$). 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. Concatenation extends to sequences: if $a \in \Sigma^*$ is a finite sequence and $b \in S$, ab is a followed by b.

We write $a \sqsubseteq b$ if a is a prefix of or equal to b; $\sqsubseteq$ 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 $\mathbf{b} = (b_1, \ldots, b_n) \in S^n$, we write $\mathbf{a} \sqsubseteq \mathbf{b}$ iff $a_1 \sqsubseteq b_1, \ldots, a_n \sqsubseteq b_n$. Juxtaposition is elementwise concatenation: $\mathbf{ab} = (a_1b_1, \ldots, a_nb_n)$.

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 [13] 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.,

$$P(x_1x_2\cdots x_n, y_1y_2\cdots y_m) = w_1w_2\cdots w_{\min(m,n)} \tag{1}$$

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

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 and input index) to either one of the e environment channels or the output of some 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

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

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 [12] that his networks are *deterministic*: there is exactly one least $\mathbf{c}$ that satisfies (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 [13] 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 deterministically: in particular, once an actor can fire on a given rule it cannot fire on another, even if additional tokens arrive.

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 [13] 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 (3) matches the usual recursive definition of the *map* function familiar to functional programmers.

Our processes can emit an initial sequence of tokens to break initial deadlocks in loops. For example, the top multiplexers in Fig. 1 would deadlock without the initial token provided. 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 (3) to the R and f in (5) gives the P function in (1). In Fig. 1, the equality tester (=), less-than (<), and subtractor (−) actors are each unit-rate.

Our second building block is the *mux* actor (Fig. 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* (Fig. 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

The determinism of Kahn networks means they cannot model nondeterministic merge processes. Although we desire deterministic I/O, we also want to use merge actors judiciously, as they provide a mechanism for sharing hardware resources (explained below).

We adopt Broy's [2] solution: the behavior of a process is a set of continuous functions. Most processes have only a single function in their set, but the set for a merge actor includes one function for every possible interleaving of its inputs. We think of a nondeterministic merge as a mux with a hidden select input, so the behavior of a network is any fixed point consistent with some select input sequence for each merge.

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 their functional behavior. Specifically, multiple actors may be chained together directly for speed or buffers may be added to break frequency-robbing critical paths through pipelining.

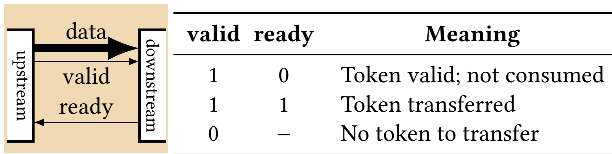
To implement a dataflow network, each dataflow actor becomes a block of logic with handshaking communication ports, one for each input and output. Each channel in the network becomes a small communication network of wires potentially augmented with fork and buffering circuity (§4). Aside from the need 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. Due to space, we do not include formal proofs that show each circuit faithfully implements its specification.

We only 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. [18]). Our framework could support additional actor types, but their design is outside the scope of this paper.

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 [4] and elastic circuits [6] shown in Fig. 2. This is a bundled data protocol in which the *valid* bit indicates a token is present on the *data* wires.



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

Figure 2: A point-to-point link and its protocol [3]: *valid* indicates the data lines carry a token; *ready* indicates downstream is willing to consume a token.

The downstream block sends the *ready* signal upstream to indicate it is able to consume a token being proffered by the upstream block.

A token is transferred from the upstream block to the downstream block when both *valid* and *ready* are asserted.

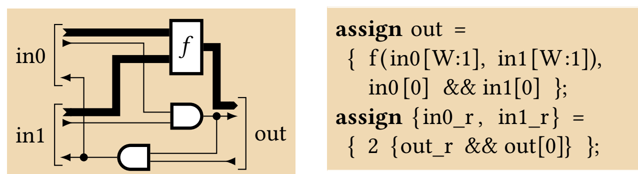
We provide a fragment of synthesizable RTL SystemVerilog for each block. To represent, say, an 8-bit channel *c*, we use a nine-bit vector *c* for data (c[8:1]) and *valid* (c[0]), and a wire named *c_r* for *ready*. In our schematics, we label all wires of this port just *c*.

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 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 §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, the flow-control network can be scheduled statically: the *valid* network can be computed first (it is acyclic, with inputs from data buffers and the environment) followed by the *ready* network, which may take inputs from the *valid* network, control buffers, and the environment.

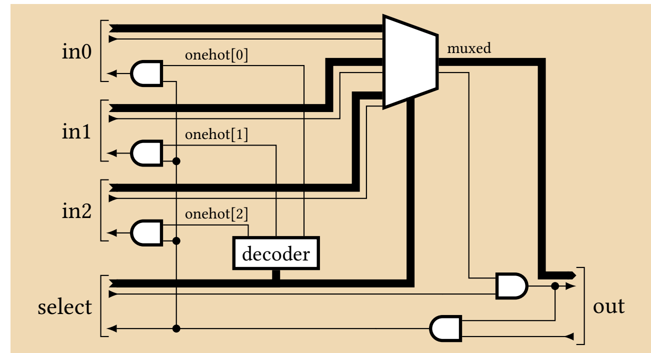
3.2 Unit-Rate Actors

Fig. 3 shows how we implement single-output unit-rate actors such as the two-input adder. This actor waits for a valid token on both its inputs before asserting its output is valid. It 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 two-input circuit of Fig. 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.




```
assign out =
  { f(in0[W:1], in1[W:1]),
    in0[0] && in1[0] };
assign {in0_r, in1_r} =
  { 2 {out_r && out[0]} };
```

Figure 3: A unit-rate actor: computes f of two W-bit inputs




```
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;
```

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

3.3 Mux and Demux Actors

Mux and demux actors are not unit-rate actors. 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.

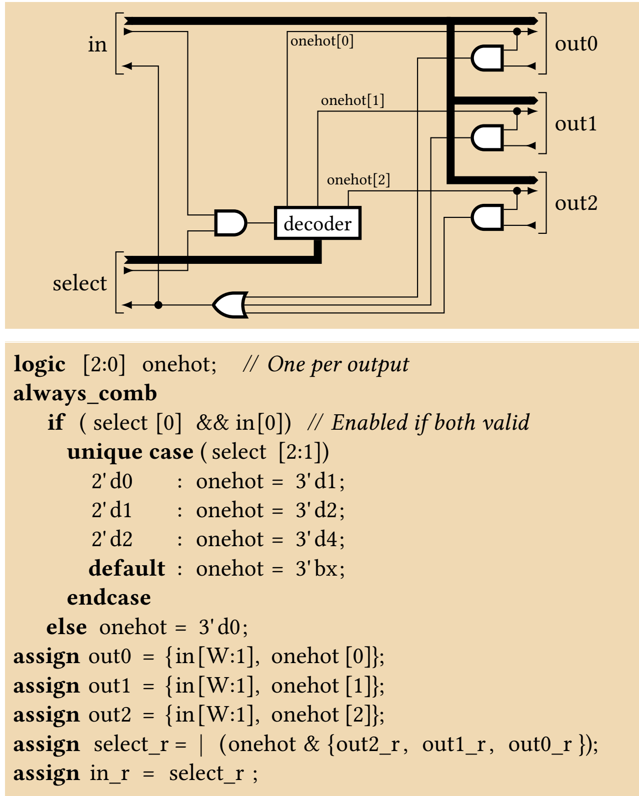
The demux is complementary: it directs an input token to a single, specific output depending on the value of a select token. Both the select and input tokens must be valid before a token is proffered on the selected output; that output must be ready before the two tokens are consumed.

Fig. 4 shows our implementation of a three-input multiplexer that routes W-bit tokens. The local signal *muxed* comes from one of the input ports according to the value of the *select* token, which is also decoded to generate the *onehot* vector.

The downstream *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 downstream output is valid and ready.

Fig. 5 shows a three-output demultiplexer. The datapath copies input data to all outputs. If both the *in* port and the *select* port have valid tokens, the one-hot decoder uses the value of the *select* token to indicate exactly one of the output ports has a valid token. Both inputs are consumed if the selected output is ready.

Note that each actor is implemented as a "stock" datapath for a combinational function, a multiplexer, and fan-out augmented with flow-control logic. Our technique therefore does not introduce additional delay into the data network.




```
logic [2:0] onehot; // One per output
always_comb
  if ( select [0] && in[0]) // Enabled if both 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 ;
```

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

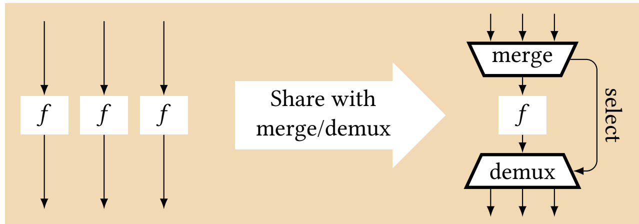
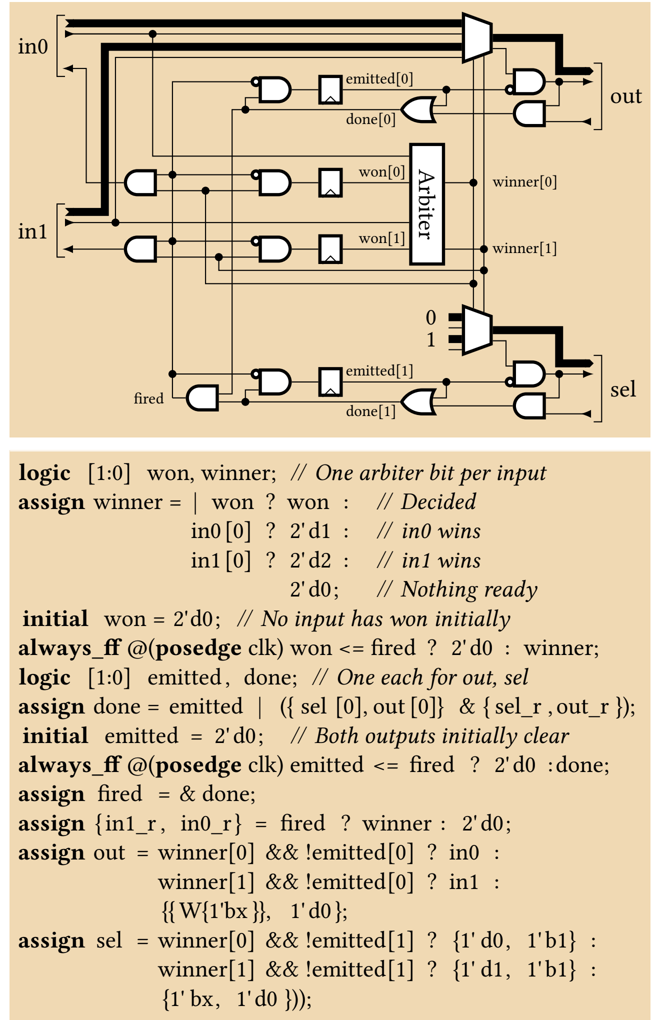


Figure 6: A merge used to share a unit-rate subnetwork.

3.4 Merge Actors

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

Fig. 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.




```
logic [1:0] won, winner; // One arbiter bit per input
assign winner = | won ? won :   // Decided
                in0 [0] ? 2'd1 : // in0 wins
                in1 [0] ? 2'd2 : // in1 wins
                          2'd0;  // Nothing ready
initial won = 2'd0; // No input has won initially
always_ff @(posedge clk) won <= fired ? 2'd0 : winner;
logic [1:0] emitted, done; // One each for out, sel
assign done = emitted | ({ sel [0], out [0]} & { sel_r , out_r });
initial emitted = 2'd0; // Both outputs initially clear
always_ff @(posedge clk) emitted <= fired ? 2'd0 :done;
assign fired = & done;
assign {in1_r, in0_r} = fired ? winner : 2'd0;
assign out = winner[0] && !emitted[0] ? in0 :
             winner[1] && !emitted[0] ? in1 :
             {{W{1'bx }}, 1'd0 };
assign sel = winner[0] && !emitted[1] ? {1'd0, 1'b1} :
             winner[1] && !emitted[1] ? {1'd1, 1'b1} :
             {1'bx, 1'd0 }));
```

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

Fig. 7 shows a two-input nondeterministic merge actor, which interleaves tokens on inputs *in0* and *in1* onto the *out* port and reports how the interleaving was done on the *sel* port. As we described in §2.4, Broy [2] 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 Fig. 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 §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 *emitted* 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 selects a valid input and declares it the winner through the one-hot *winner* vector. This vector controls the multiplexers that route the winning input to *out* and its identity to *sel*.

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

When an output is not ready, it sets the *emitted* bit for the other output, suppressing that output's *valid* signal in the next cycle. Furthermore, because *fired* is not asserted, the *winner* 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 *winner* 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.

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 Fig. 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 Kahn network and on channels with initial tokens (see §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*.

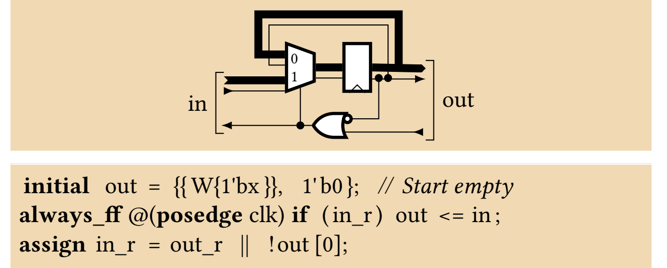
Choosing an optimal channel implementation is outside the scope of this paper. However, we can correctly implement any channel in our specification as a single-source, feed-forward network comprised of forks, buffers, and point-to-point connections.

Below, we describe how we implement buffers and forks.

4.1 Data and Control Buffers

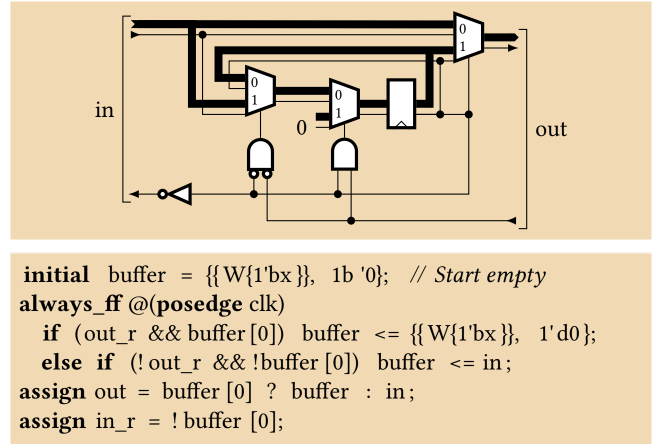
We provide two buffer types: a data buffer breaks a combinational path (or cycle) on the data/valid network; a control buffer does so on the ready network. Each type of buffer can hold a single data token, but their implementations differ.

A data buffer (Fig. 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.




```
initial out = {{W{1'bx}}, 1'b0}; // Start empty
always_ff @(posedge clk) if (in_r) out <= in;
assign in_r = out_r || !out[0];
```

Figure 8: A data buffer, after Cao et al. [3]. This pipeline register breaks a combinational path in the data/valid network.




```
initial buffer = {{W{1'bx}}, 1b'0}; // Start empty
always_ff @(posedge clk)
  if (out_r && buffer[0]) buffer <= {{W{1'bx}}, 1'd0};
  else if (!out_r && !buffer[0]) buffer <= in;
assign out = buffer[0] ? buffer : in;
assign in_r = !buffer[0];
```

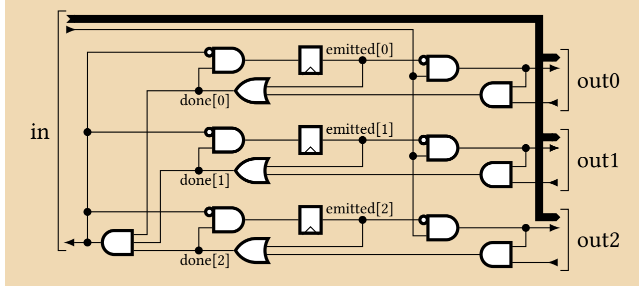
Figure 9: A control buffer, after Cao et al. [3]. This breaks a combinational path in the (upstream) *ready* network.

The control buffer in Fig. 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. In normal operation, the buffer in Fig. 9 is 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, the buffer remains empty (holds its previous contents); however, if *out_r* goes false, 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, they behave like the latency-insensitive relay stations of Li et al. [15].




```
logic  [2:0]  emitted, done; // One per output
initial  emitted = 3'd0;     // All channels initially clear
assign out0 = { in[W:1], in[0] && !emitted[0] };
assign out1 = { in[W:1], in[0] && !emitted[1] };
assign out2 = { in[W:1], in[0] && !emitted[2] };
assign done = emitted | ({out2[0], out1[0], out0[0]} &
                          {out2_r,  out1_r,  out0_r });
assign in_r = & done;
always_ff @(posedge clk) emitted <= in_r ? 3'd0 : done;
```

Figure 10: A three-way fork. An output port's *emitted* 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.

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 in the generated circuit.

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 implies a path from *ready* to *valid*. A block that considered *ready* inputs to determine whether to fire would not be compositional.

Our implementation of fork avoids combinational cycles by using one flip-flop per output. It allows a valid token to pass through a fork and be consumed downstream *before* it is consumed upstream.

Fig. 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, 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, the state is unchanged. If an input token arrives it is proffered on all downstream ports. If all consumers are ready, *done* is all ones, the upstream *ready* is asserted, and the *emitted* flip-flops remain cleared.

If any consumer is not ready, 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 §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 (3), when a process finds tokens on its input sequences that match a firing rule **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 **t**, which skips the tokens in the firing rule **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, 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 $\ldots, c_{t-1}, c_t, c_{t+1}, \ldots$, we might observe

data:	⋯	c_{t-1}	c_{t-1}	c_{t-1}	x	c_t	c_t	c_{t+1}	⋯
valid:	⋯	1	1	1	0	1	1	1	⋯
ready:	⋯	0	0	1	x	0	1	0	⋯

Here, the highlighted columns denote token-transfer cycles. The environmental inputs must follow this protocol.

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** preserves the invariant. According to the logic in its schematic, if all its inputs are valid, 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, the inputs are also made ready, indicating the input tokens have been consumed.

For the **multiplexer**, 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, the output carries the proper value (next token in sequence) and is set valid. If, additionally, the output is ready, both the *select* input and the selected input are made ready and no others.

For the **demultiplexer**, 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, only then are both the input and select inputs marked ready.

The **nondeterministic merge** block 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, *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, *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, *fired* will be asserted, the winning input will be made ready, the registers cleared, and the process repeats.

Buffers are the only blocks that hold tokens. Consider when the **data buffer** is empty. The output is invalid and the *ready* output is asserted. If a valid token is proffered, 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, 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, it will be stored in the buffer.

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

The **fork** block 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, 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 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 manually and using the compiler of Townsend et al. [18]) buffered them both

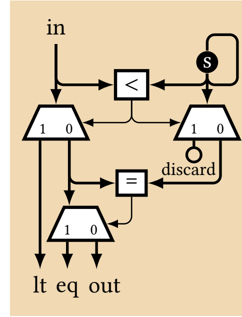


Figure 11: The splitter component of a Conveyor. This network partitions tokens arriving on the input stream *in* by comparing each input value against a "split" value: the initial token *s*. Each input token is sent out on one of three ports depending on whether it is less than (*lt*), equal to (*eq*), or greater than (*out*) the split value.

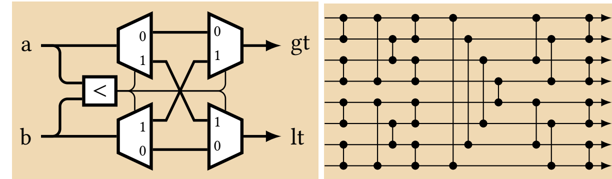


Figure 12: The network on the left routes the larger of two input tokens to the top output and the smaller to the bottom. These are the vertical lines in the eight-element bitonic sorting network on the right (after Cormen et al. [7]).

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.

As part of this work, we developed a compiler that takes a low-level textual dataflow language that specifies the topology of a network composed of blocks, buffers, forks, and point-to-point channels and generates synthesizable SystemVerilog for each block following the templates described in §3 and §4.

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.

6.1 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. [18].

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

Conveyor. This network performs range partitioning [20]. The design chains n splitters (Fig. 11) 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.

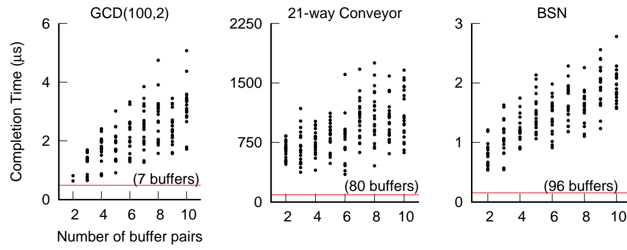


Figure 13: Completion times under random buffer placement. Horizontal lines labeled with a buffer count indicate the completion time of the best manual design.

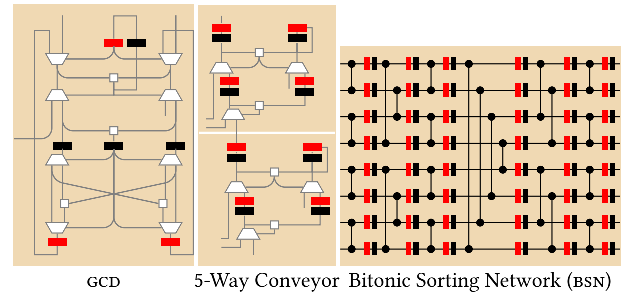


Figure 14: Buffering our networks. Red bars represent control buffers; black for data. Each cycle requires both buffers.

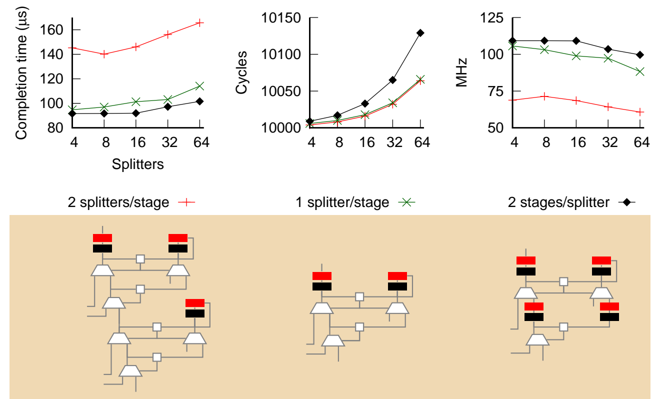


Figure 15: Three Conveyor pipelining strategies: doubling the size has only a small effect on total completion time

Bitonic Sorting Network (BSN). This sorts a fixed number of values with two-input comparators that operate in parallel. Fig. 12 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 ten 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. 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.

6.2 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. Fig. 13 shows the completion time of each of these implementations in microseconds: cycles divided by maximum frequency (MHz).

6.3 Manual Buffer Allocation

In Fig. 13, 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.

Fig. 14 depicts our best manual buffer placements. Each black bar represents a data buffer; red represents a control buffer. For space, we only depict two representative splitters (of ten) in the Conveyor.

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 (103 MHz for BSN; 109 MHz 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 Fig. 14 improved the frequency from 79 MHz to 109 MHz.

6.4 Pipelining the Conveyor

Over Conveyors with 4 to 64 splitters, we experimented with the three pipelining strategies shown in Fig. 15: 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.

6.5 Memory in Dataflow Networks

We plan 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. [18] 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 runs faster with less logic but more memory because its (two-pointer) tree objects are wider than (one-pointer) list objects: it completes in 94.8 μs, operates at 54 MHz, and uses 3,330 ALMs and 8.7 kB of memory, while MergeSort takes 114.8 μs, running at 49 MHz with 3,592 ALMs and 5.9 kB. These are meant to demonstrate that our formalism readily accommodates memory and are not high-performance sorters.

7 RELATED WORK

Surprisingly little has been written about synthesizing hardware from a dataflow model as rich as ours. Tripakis et al. [19] surveys a number of dataflow-to-hardware projects, but most have focused on statically schedulable models such as SDF that do not support actors such as mux and demux that make data-dependent decisions. For example, the LID work of Carloni et al. [5] was inspirational for us but only considers unit-rate actors.

Perhaps closest in spirit to our work is that of Janneck et al. around CAL [10]: a rich, functional-inspired language for expressing dataflow process actors and networks. Janneck et al. [1, 11] 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. [17] synthesize hybrid hardware/software systems from CAL.

The Elastic Systems of Cortadella et al. [8, 9] 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.

Possignolo et al. [16] also consider token/handshaking pipelines for processor design. They use Colored Petri Nets to model throughput, something we may be able to adopt.

8 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] 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)*. The Institute of Electrical and Electronics Engineers (IEEE), Trieste, Italy, 750–755. https://doi.org/10.1109/ISPA.2013.6703837

[2] Manfred Broy. 1988. Nondeterministic Data Flow Programs: How To Avoid the Merge Anomaly. *Science of Computer Programming* 10, 1 (Feb. 1988), 65–85. https://doi.org/10.1016/0167-6423(88)90016-0

[3] 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)*. ACM, Austin, Texas, 179–187.

[4] Luca P. Carloni. 2006. The Role of Back-Pressure in Implementing Latency-Insensitive Systems. *Electronic Notes in Theoretical Computer Science* 146, 2 (2006), 61–80.

[5] Luca P. Carloni, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli. 2001. Theory of Latency-Insensitive Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20, 9 (Sept. 2001), 1059–1076. http://www.gigascale.org/pubs/430.html

[6] Josep Carmona, Jordi Cortadella, Mike Kishinevsky, and Alexander Taubin. 2009. Elastic Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 10 (Oct 2009), 1437–1455. https://doi.org/10.1109/TCAD.2009.2030436

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

[8] 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)*. The Institute of Electrical and Electronics Engineers (IEEE), Grenoble, France, 149–158. https://doi.org/10.1109/MEMCOD.2010.5558639

[9] Jordi Cortadella, Mike Kishinevsky, and Bill Grundmann. 2006. SELF: Specification and design of synchronous elastic circuits. In *ACM Intl. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*. San Jose, California, 6.

[10] 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. http://www.eecs.berkeley.edu/Pubs/TechRpts/2003/4186.html

[11] 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. *Journal of Signal Processing Systems* 63, 2 (July 2009), 241–249. https://doi.org/10.1007/s11265-009-0397-5

[12] 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.

[13] 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.

[14] Edward A. Lee and Thomas M. Parks. 1995. Dataflow Process Networks. *Proc. IEEE* 83, 5 (May 1995), 773–801. http://ptolemy.eecs.berkeley.edu/papers/processNets

[15] 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)*. IEEE, Nice, France, 13–22.

[16] 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)*. The Institute of Electrical and Electronics Engineers (IEEE), Scottsdale, Arizona, 233–240. https://doi.org/10.1109/ICCD.2016.7753285

[17] 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)*. ECSI, Sophia Antipolis, France, 127–134.

[18] Richard Townsend, Martha A. Kim, and Stephen A. Edwards. 2017. From Functional Programs to Pipelined Dataflow Circuits. In *Proceedings of Compiler Construction (CC)*. ACM, Austin, Texas, 76–86. https://doi.org/10.1145/3033019.3033027

[19] 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 *Embedded Computer Systems: Architectures, Modeling, and Simulation (Proceedings of SAMOS Workshop)*. IEEE, Samos, Greece, 51–58. https://doi.org/10.1109/SAMOS.2014.6893194

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