PAClib Reference Guide

Version 1.4 (May 15, 2014)

Contents

1 Introduction

PAClib (Pipelined Architecture Composers library) is a product from Bluespec, Inc. for high-level modeling and implementation of pipelined architectures (algorithm and datapath designs) using BSV (Bluespec SystemVerilog). PAClib is a source-code library, and is therefore synthesizable (since everything in BSV is synthesizable). Thus, even high-level models can be emulated at high speeds on FPGA platforms.

PAClib consists of a set of standard plug-and-play pipeline building blocks, user-parametrized by computational functions, structures, buffering and data types. Providing 100% transparency and control of architectures, PAClib enables the rapid creation of a single algorithm specification that can generate many different micro-architectures for rapid architectural exploration—and correct control-logic is automatically generated for each micro-architecture. PAClib can be seamlessly integrated and mix-and-matched with high-level complex control, so that designs of any size and complexity can be specified, and considerations such as memory accesses and data movement can be easily and efficiently incorporated in the same design.

Unlike C/C++/SystemC approaches to high-level synthesis, the PAClib approach allows the designer to control architecture precisely and predictably, and therefore to converge quickly to results meeting price, performance and power targets.

The target audience for PAClib is primarily DSP/algorithm designers who want a fast and effective way of rendering mathematical algorithm specs into hardware. In addition, they may also want a way to express multiple alternative architectures in a unified way, because they are targeting various platforms with different requirements for performance, area, power, etc. Additionally, PAClib may also be used for various ad hoc pipelines that occur across the spectrum of designs (not necessarily DMA-oriented).

PAClib uses features in BSV and in the world's most advanced programming languages such as atomic transactions, higher-order functions, extreme parametrization, polymorphism, and user-extensible overloading. Because of the expressive power of BSV, PAClib is implemented entirely in BSV source code, and is thus fully extensible, customizable and synthesizable.

This document is a reference guide for PAClib, and is not a tutorial either on PAClib or on BSV. For a tutorial introduction to PAClib, please download the free technical whitepaper entitled, "High-level 'plug-and-play' specification, modeling and synthesis of pipelined architectures with Bluespec's PAClib", which includes a case study showing IFFT in 100 lines of code, generating 24 micro-architectures for FPGAs and ASICs. The whitepaper is available at http://www.bluespec.com/algorithmic-synthesis-paclib.htm. For a tutorial introduction to BSV, please contact Bluespec, Inc. or peruse the training material at www.bluespec.com.

2 General principles of PAClib

Many computations begin as mathematical specifications. This is certainly true of many signal-processing applications in wireless and multimedia. PAClib narrows the gap to hardware implementation by taking a mathematical (or functional) approach to expressing pipeline structures. A module with interface type `Pipe#(a,b)` is considered to be an implementation of a function f from inputs of type `a` to outputs of type `b`. In other words, for each input x of type `a`, after some pipeline delay the module yields an output y such that $y = f(x)$. A sequence of inputs $x_1, x_2, ...$ is transformed into a series of outputs $y_1, y_2, ... = f(x_1), f(x_2), ...$

In mathematics we have the concept of functional composition, $f \circ g$, where $f \circ g(x) = f(g(x))$. In PAClib this directly corresponds to cascading two pipes by connecting the output of the pipe representing g to the input of the pipe representing f. The `mkCompose` module constructor in Section A.8 describes exactly such a composition. The overall philosophy in PAClib is to compose pipeline components in the same way you compose functions in mathematics. Thus, one composes complex pipeline structures by starting with primitive functions and encapsulating them as primitive pipelines, composing them to form more complex pipelines which can themselves be composed into even more complex pipelines.

Many signal-processing applications involve vector transformations where vectors may be rendered either in space (arrays) or in time (streams or sequences). Mathematical concepts such as "mapping" (applying a function to each element of a vector to produce a vector of results) and "reduction" (combining items from a sequence using a binary combining operator) have direct counterparts in PAClib[1].

This natural rendering of mathematical specifications into hardware architectures relies on the expressive power of BSV, notably polymorphism, overloading, higher-order functions, and so on. But, most crucially, it relies on the power of atomic transactions that are at the heart of BSV. Many of the structures described below, in particular the ones that involve temporal iteration (loops), require quite complex control logic that may, in turn, vary depending on the loop body. High-level synthesis from atomic transactions automates this control logic. Finally, because PAClib is a source-code library, it is truly extensible and customizable for situations that may not have been anticipated in the design of PAClib.

In almost all the PAClib functions, the data types flowing through the pipelines are expected to be in the `Bits#(n)` typeclass because, of course, they are dynamic types represented in hardware. Accordingly, for brevity we do not specify this proviso explicitly in any of the descriptions that follow.

In all the PAClib functions, every attempt has been made to avoid "dead cycles", that is, to avoid introducing bubbles into the pipelines. In addition, every attempt has been made to avoid introducing unnecessary state, so that the hardware cost of each constructor is obvious. Wherever non-trivial extra state is necessary, it is described explicitly in the text accompanying the constructor specification.

[1] Others have also recognized the power of functional abstractions for high-performance parallel computing. See, for example, *MapReduce: Simplified Data Processing on Large Clusters*, Jeffrey Dean and Sanjay Ghemawat, Proc. 6th. Symp. on Operating System Design and Implementation (OSDI), San Francisco, CA, December, 2004. http://labs.google.com/papers/mapreduce.html

A PAClib.bsv

This section describes the BSV data types, interfaces, modules and functions which are provided by the PAClib package. To access the library, import the PAClib package:

```
import PAClib :: * ;
```

This will, in turn, import the packages `FIFO`, `FIFOF`, `SpecialFIFOs`, `GetPut`, `ClientServer`, `Connectable`, `Vector`, and `CompletionBuffer`.

A.1 Data Types

The interface type `PipeOut` represents the output of a pipeline module, and is basically the same as the output part of a `FIFOF` interface:

```
interface PipeOut #(type a);
   method a       first    ();
   method Action  deq      ();
   method Bool    notEmpty ();
endinterface
```

As in `FIFOF`, the `first` method is a non-destructive examination of the current output yielded by the pipe (if there is one available), the `deq` method allows the data to advance by discarding the current output, and `notEmpty` method allows testing whether there is a current output available. As in `FIFOF`, the `first` and `deq` methods have implicit conditions so that they can be invoked only if there is a current output, so it is not necessary to test `notEmpty`. One usually needs the `notEmpty` test only in situations where some other useful work can be done while there is no current output available from the pipe.

The type `Pipe` represents a constructor for a pipeline module.

```
typedef (function Module #(PipeOut #(b)) mkPipeComponent (PipeOut #(a) ifc))
        Pipe#(type a, type b);
```

This is a function whose argument is the input interface of the pipeline module (type `PipeOut#(a)`) and which produces a module with the output interface (type `PipeOut#(b)`). This is illustrated in Fig. 1. Thus, module composition (cascading) corresponds directly to function composition in the

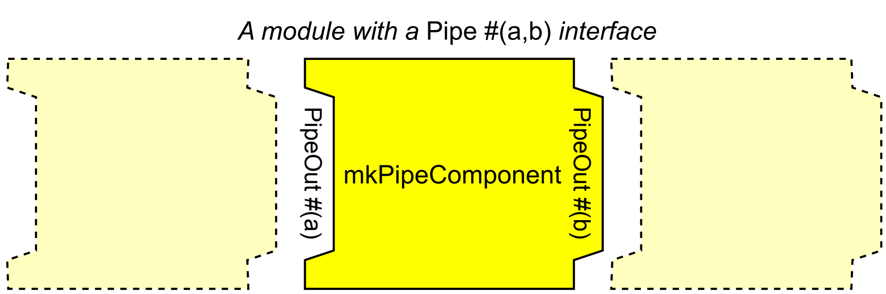

Figure 1: A module with a `Pipe` interface has a `PipeOut` input interface and an `PipeOut` output interface.

source code, that is, we will apply a function like `mkPipeComponent` to an existing `PipeOut` interface (the upstream module), and this will yield a module with a `PipeOut` interface to which, in turn, we will apply another `mkPipeComponent`-like function (the downstream module), and so on.

The interface type PipeOut is an instance of the ToGet typeclass. Therefore, the following overloaded function is available:

toGet	Convert a PipeOut interface to a Get interface.
	`instance ToGet #(PipeOut #(a), a);`

This enables, for example, using mkConnection to connect a PipeOut with some other Put interface (see Connectables package in BSV libraries).

A.2 Utility functions

The first two utilities enable you to view FIFOF interfaces of existing modules as pipeline interfaces.

f_FIFOF_to_PipeOut	Convert (the output part of) a FIFOF interface to a PipeOut interface.
	`function PipeOut #(a)  f_FIFOF_to_PipeOut  (FIFOF #(a) fifof);`

mkFIFOF_to_Pipe	Create a pipeline module from a FIFOF interface.
	`function Pipe #(a,a) mkFIFOF_to_Pipe (FIFOF #(a) fifof);`
	`module mkFIFOF_to_Pipe #(FIFOF #(a) fifof, PipeOut #(a) po_in) (PipeOut #(a));`

```
module mkFIFOF_to_Pipe
               #(FIFOF #(a)    fifof,
                 PipeOut #(a)  po_in)
               (PipeOut #(a));
```

This effectively views an existing FIFOF module as a Pipe module. If you want to simply create a new Pipe module containing a FIFOF buffer, please see Section A.4.

mkPipe_to_Server	Create a Server module from a pipeline constructor.
	`function Module #(Server #(a, b) mkPipe_to_Server (Pipe #(a, b) mkP);`
	`module mkPipe_to_Server #(Pipe #(a, b) pipe) (Server #(a, b));`

```
function Module #(Server #(a, b)
         mkPipe_to_Server (Pipe #(a, b) mkP);
```

```
module mkPipe_to_Server
       #(Pipe #(a, b) pipe)
       (Server #(a, b));
```

This function is typically applied to the top-level pipeline constructor of a complex pipeline, and "caps it off" into a module with a conventional Server interface, i.e., with a conventional put method to inject inputs and a conventional get method to retrieve outputs. It introduces one level of FIFOF buffering (and therefore one tick of latency) in front of mkP. This is illustrated in Fig. 2. Recall that a Server interface is frequently-used BSV library interface with facilities to mkConnection with Client interfaces, and more.

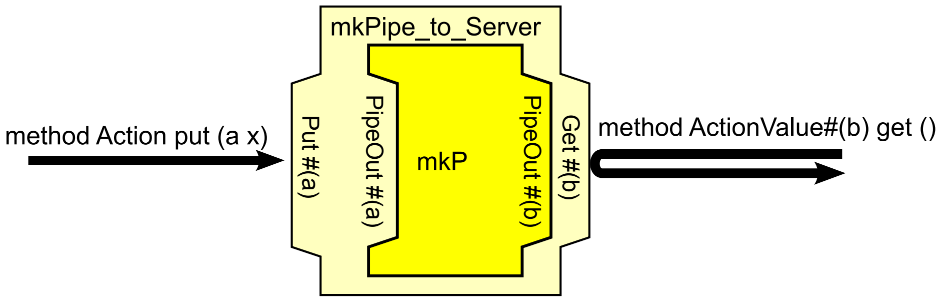

Figure 2: `mkPipe_to_Server` "caps off" a `Pipe` module into a `Server` module.

A.3 Pipeline Sources and Sinks

The constructors below are illustrated in Fig. 3.

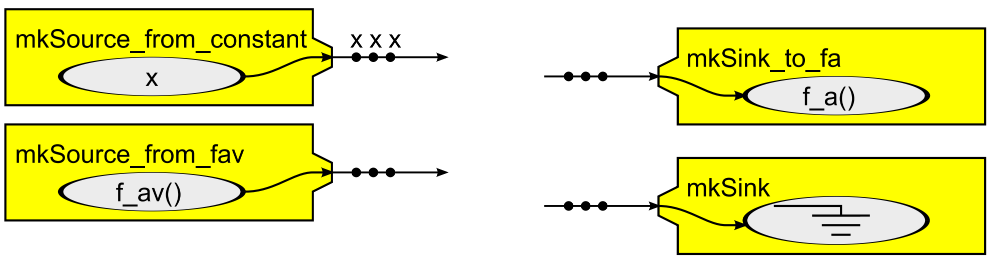

Figure 3: Sources and Sinks.

`mkSource_from_constant`	Create a module with a `PipeOut` interface that repeatedly yields a given constant `x`.
	<pre>module mkSource_from_constant #(a x) (PipeOut #(a));</pre>

`mkSource_from_fav`	Create a module with a `PipeOut` interface that yields values from repeated invocations of an ActionValue function `f_av()`.
	<pre>module mkSource_from_fav #(ActionValue #(a) f_av) (PipeOut #(a));</pre>

For example, `f_av` may contain a counter yielding successive values, or it may read from a memory or an input port, or (in simulation) may read from a file or the console.

`mkSink`	Create a module with an input `PipeOut` interface that accepts values and discards them.
	<pre>module mkSink #(PipeOut #(a) po_in) (Empty);</pre>

mkSink_to_fa	Create a module with an input PipeOut interface that accepts values and sends them into an Action function f_a().	
	`module mkSink_to_fa #(function Action f_a (a x), PipeOut #(a) po_in) (Empty);`	

For example, f_a may write to a memory, or to an output port, or (in simulation) may write to the console or a file.

A.4 Pipeline buffers

Below, mkBuffer and mkBuffer_n contain FIFO buffers, and so the input and output interfaces are asynchronous, that is, putting a value into the input PipeOut interface need not be simultaneous with taking a value from the output PipeOut interface.

On the other hand, mkSynchBuffer and mkSynchBuffer_n contain register buffers, and the input and output interfaces are synchronous, that is, putting a value into the input PipeOut interface is always simultaneous with taking a value from the output PipeOut interface.

The constructors below are illustrated in Fig. 4.

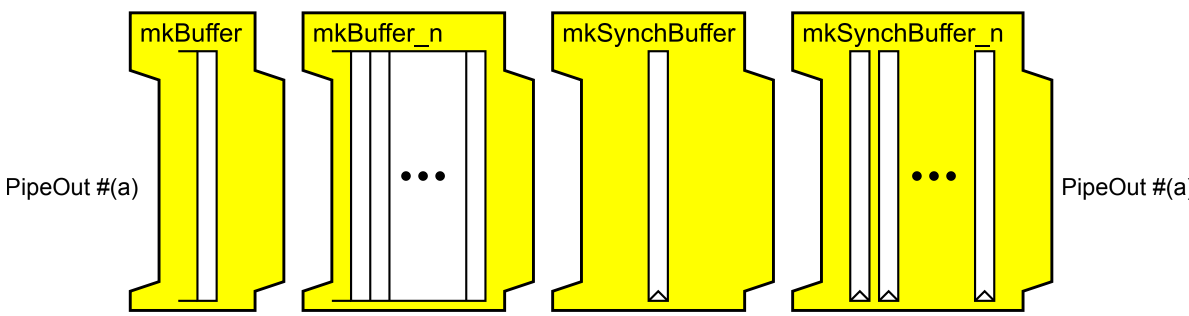

Figure 4: Buffers, asynchronous (FIFO-like) and synchronous (register-like).

mkBuffer	Create a Pipe module containing a FIFO buffer.
	`function Pipe #(a, a) mkBuffer ();`
	`module mkBuffer #(PipeOut #(a) po_in) (PipeOut #(a));`

This encapsulates a mkLFIFOF into a Pipe module.

`mkBuffer_n`	Create a `Pipe` module containing an n-deep FIFO buffer.	
	`function Pipe #(a, a) mkBuffer_n (Integer n);`	
	`module mkBuffer_n #(Integer n, PipeOut #(a) po_in) (PipeOut #(a));`	

This encapsulates a `mkSizedFIFOF` into a `Pipe` module. Note: whereas a composition of `n` `mkBuffer`s would have a latency of at least `n` (one tick through each buffer), a `mkBuffer_n`'s latency just depends on how many items are in the FIFO, and can be as low as 1 (when the buffer is empty).

`mkSynchBuffer`	Create a `Pipe` module containing a synchronous (register) buffer with initial contents `init_value`.
	`function Pipe #(a, a) mkSynchBuffer (a init_value);`
	`module mkSynchBuffer #(a init_value, PipeOut #(a) po_in) (PipeOut #(a));`

This encapsulates a `mkReg` into a `Pipe` module.

`mkSynchBuffer_n`	Create a `Pipe` module containing n-stages of synchronous (register) buffers, all with initial contents `init_value`.
	`function Pipe #(a, a) mkSynchBuffer_n (a init_value);`
	`module mkSynchBuffer_n #(Integer n, a init_value, PipeOut #(a) po_in) (PipeOut #(a));`

This is equivalent to composing `n` copies of `mkSynchBuffer` into a linear pipe. Thus, `mkSynchBuffer_n` acts as an `n`-stage shift register, and has a fixed latency of `n`. In particular `mkSynchBuffer_n(1)` is the same as `mkSynchBuffer`.

A.5 Wrapping combinational and Action functions into `Pipe` modules

`mkFn_to_Pipe`	Create a `Pipe` module where, for each input `x`, the output is `fn(x)`.
	`function Pipe #(a, b) mkFn_to_Pipe (function b fn (a x));`
	`module mkFn_to_Pipe #(function b fn (a x), PipeOut #(a) po_in) (PipeOut #(b));`

Note that the input type `a` and output type `b` may be different.

`mkFn_to_Pipe_Buffered`	Create a `Pipe` module where, for each input `x`, the output is `fn(x)`. The boolean parameters control whether a FIFO buffer is placed before and after the function, respectively.
	`function Pipe #(a, b) mkFn_to_Pipe_Buffered (Bool param_buf_before, function b fn (a x), Bool param_buf_after);`
	`module mkFn_to_Pipe_Buffered #(Bool param_buf_before, function b fn (a x), Bool param_buf_after, PipeOut #(a) po_in) (PipeOut #(b))`

This is illustrated in Fig. 5.

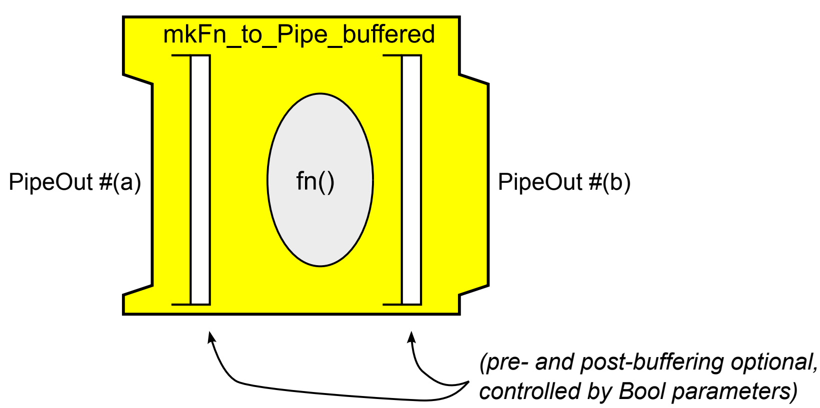

Figure 5: `mkFn_to_Pipe_Buffered` structure.

Note: `mkFn_to_Pipe_Buffered (False, fn, False)` is equivalent to `mkFn_to_Pipe(fn)`.

mkFn_to_Pipe_SynchBuffered	
	Create a `Pipe` module where, for each input `x`, the output is `fn(x)`. The `Maybe` parameters control whether a register buffer is placed before and after the function, respectively.
	`function Pipe #(a, b) mkFn_to_Pipe_SynchBuffered (Maybe #(a) param_buf_before, function b fn (a x), Maybe #(b) param_buf_after);`
	`module mkFn_to_Pipe_SynchBuffered #(Maybe #(a) param_buf_before, function b fn (a x), Maybe #(b) param_buf_after, PipeOut #(a) po_in) (PipeOut #(b));`

If the first `Maybe` parameter is `Valid` with value `init_a`, then a register buffer is placed before the function with initial contents `init_a`. Similarly, if the second `Maybe` parameter is `Valid` with value `init_b`, then a register buffer is placed after the function with initial contents `init_b`.

Note: `mkFn_to_Pipe_SynchBuffered (False, fn, False)` is equivalent to `mkFn_to_Pipe (fn)`.

mkRetimedPipelineFn	Wrap a function in a `Pipe` module and add n stages of registers to allow for retiming
	`module mkRetimedPipelineFn (function b func (a x), Integer stages PipeOut#(a) pin, PipeOut#(b) ifc);`

mkTap	Create a `Pipe` module that applies an Action function `tap_fn_a()` to each value flowing through.
	`function Pipe #(a, a) mkTap (function Action tap_fn_a (a x));`
	`module mkTap #(function Action tap_fn_a (a x), PipeOut #(a) po_in) (PipeOut #(a));`

For example, `tap_fn_a` may write a trace log to an I/O port, memory or (in simulation) a file. This is illustrated in Fig. 6. This module does not add any state (no extra latency).

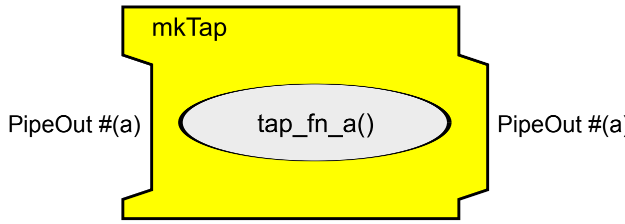

Figure 6: `mkTap` enables performing an Action on values passing through.

`mkReplicateFn`	Create a `Pipe` module that iteratively applies a function `fn` to each value flowing through.
	`function Pipe #(a, a) mkTap (function Action tap_fn_a (a x));`
	`module mkReplicateFn #(Integer apply_count,` `                        function b fn (a x, UInt#(n) cnt),` `                        PipeOut #(a) po_in)` `     (PipeOut #(b));`

For each input value `x`, produce `fn(x,0)`, `fn(x,1)`, ..., `fn(x,apply_count-1)` on the output.

A.6 Funneling and Unfunneling (Serialization and Deserialization)

The following functions are for creating structures that "slice up" a vector and feed the slices sequentially (funneling or serialization), and for the inverse, that is, for accepting slices sequentially and reassembling them into vectors (unfunneling or deserialization).

One typical use is accept from, or to feed I/O ports where the port width may be narrower than the natural vector size on which we wish to operate.

Another typical use is in "resource-constrained" mapping (see A.7): we wish to apply some processing to each element of an input vector, but we cannot afford the circuit resources to perform them all in parallel; so, we funnel it down to narrower width, process it with fewer resources, and then unfunnel it up to full width

`mkFunnel`	Create a `Pipe` module that funnels each input vector into sequential slices.
	`function Pipe #(Vector #(mk, a), Vector #(m, a)) mkFunnel` `        provisos (...);`
	`module mkFunnel` `                #(PipeOut #(Vector #(mk, a)) po_in)` `                (PipeOut #(Vector #(m, a)))` `        provisos (...);`

For each input vector of size `mk`, it yields a sequence of `k` slices, where each slice is a vector of size `m`. Thus, `m`×`k`=`mk`. The "..." provisos ensure that `mk`>0, `m`>0, `m`≤`mk`, and that `mk` is a whole multiple of m. If the input vector is V, the first sequential slice represents V[0]...V[m-1]; the second sequential slice represents V[m]...V[2m-1]; and so on. This is illustrated in Fig. 7.

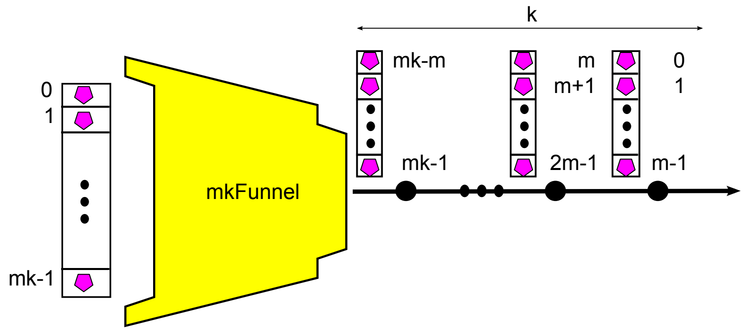

Figure 7: `mkFunnel` yields a sequence of slices of a vector

When m=mk it represents the corner case of no serialization. When m= 1 it represents the corner case of complete serialization. The pipeline has no dead cycles, that is, when the last slice V[mk-k]...V[mk-1] is yielded, it can simultaneously accept the next input vector V'.

`mkFunnel` contains a few counters for book-keeping, that is, for keeping track of which slice is next, but it does not itself buffer the vector data.

`mkFunnel_Indexed`	Create a `Pipe` module that funnels each input vector into sequential slices, with each value accompanied by its original index.
	<pre>function Pipe #(Vector #(mk, a), Vector #(m, Tuple2 #(a, UInt #(logmk)))) mkFunnel_Indexed provisos (...);</pre>
	<pre>module mkFunnel_Indexed #(PipeOut #(Vector #(mk, a)) po_in) (PipeOut #(Vector #(m, Tuple2 #(a, UInt #(logmk))))) provisos (...);</pre>

This is a generalization of `mkFunnel`—each output slice is an m vector, but here each element is tupled with its index in the original mk-vector (0..mk-1). This is useful in applications where downstream processing needs to perform different transformations depending on the index. This is illustrated in Fig. 8. If the input vector is V, the first sequential slice represents {V[0],0}...{V[m-1],m-1}; the second sequential slice represents {V[m],m}...{V[2m-1],2m-1}; and so on.

`mkFunnel_Indexed` contains a few counters for book-keeping, that is, for keeping track of which slice is next, but it does not itself buffer the vector data.

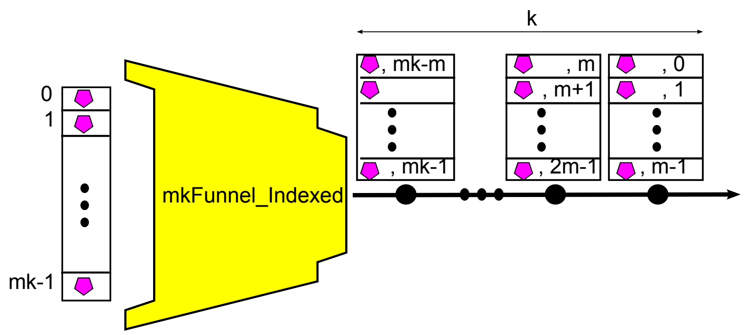

Figure 8: `mkFunnel_Indexed` yields a sequence of slices of a vector, with each element accompanied by its original index

`mkUnfunnel`	Create a `Pipe` module that unfunnels sequential slices into each output vector.
	``function Pipe #(Vector #(m, a), Vector #(mk, a)) mkUnfunnel (Bool state_if_k_is_1) provisos (...);```
	``module mkUnfunnel #(Bool state_if_k_is_1, PipeOut #(Vector #(m,a)) po_in) (PipeOut #(Vector #(mk, a))) provisos (...);```

This is the inverse of `mkFunnel`, that is, `k` sequential input slices of width `m` are reassembled into each output vector of width `mk`. The output vector V is assembled from the first sequential slice representing V[0]...V[`m`-1]; the second sequential slice representing V[`m`]...V[2`m`-1]; and so on. This is illustrated in Fig. 7. The pipeline has no dead cycles, that is, when the reassembled output vector

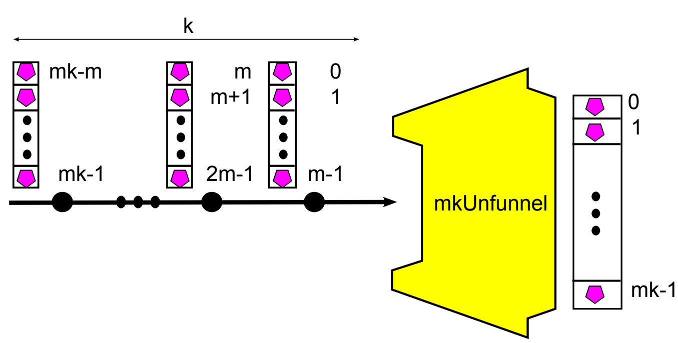

Figure 9: `mkUnfunnel` yields a sequence of slices of a vector

is yielded, it can simultaneously accept the first slice of the next vector.

`mkUnfunnel` contains a few counters for book-keeping, that is, for keeping track of which slice is next.

When m<mk, that is, when k> 1, mkUnfunnel contains a buffer for a Vector #(mk, a) value where it assembles the slices as they arrive. In this case, the boolean parameter is ignored.

In the corner case where m=mk, that is, when k= 1 and there is no unfunneling, there is no need to buffer anything. Nevertheless, when creating balanced pipelines, it may be desirable to have a level of buffering even in this corner case. The boolean parameter controls whether or not a buffer is inserted in this case.

A.7 Maps (Parallel Pointwise Applications)

Mathematically, "maps" are vectorized applications, that is, given an input vector A and a function f, the application $map(f, A)$ outputs a vector B of the same size such that $B_j = f(A_j)$ for each index j. The functions below just construct pipeline circuits corresponding to this idea.

<table>
<tr><td rowspan="3">mkMap</td><td>Create a Pipe module that maps a given Pipe module mkP over each input vector.</td></tr>
<tr><td><pre>function Pipe #(Vector #(n, a),
                Vector #(n, b))

         mkMap (Pipe #(a, b) mkP);</pre></td></tr>
<tr><td><pre>module mkMap
                #(Pipe #(a,b) mkP,
                  PipeOut #(Vector #(n, a)) po_in)

                (PipeOut #(Vector #(n, b)));</pre></td></tr>
</table>

Each element of the input vector (of type a) is sent through an instance of the pipe mkP. All the results (of type b are collected and assembled into an output vector. This is illustrated in Fig. 10. The pipe mkP can have any latency, even data-dependent latency, and in fact the different instances

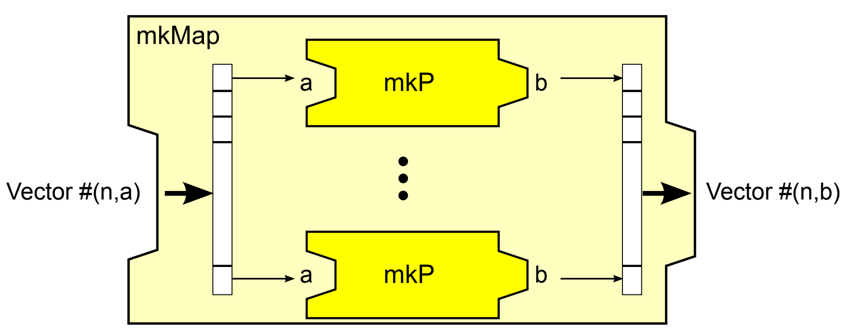

Figure 10: mkMap processes each vector element through a separate instance of mkP

of mkP do not have to have the same latency.

Special case: Combinational Maps

The above mkMap constructor assumes a component mkP that can have arbitrary latency, data-dependent latency, and indeed unequal latencies for the different indexes. In the special case that the mapped function is just a combinational function `fn`, the mapping structure can be built with the following expression:

```
mkFn_to_Pipe (map (fn));
```

Here, `map` is the BSV library vector mapping function; `map(fn)` represents the desired vector-to-vector function, and `mkFn_to_Pipe`, described earlier, simply encapsulates this into a `Pipe` module. This is slightly cheaper in hardware than the functionally equivalent `mkMap (mkFn_to_Pipe (fn))`.

<table>
<tr><td colspan="2"><code>mkMap_with_funnel_indexed</code></td></tr>
<tr><td></td><td>Create a <code>Pipe</code> module that maps a given <code>Pipe</code> module <code>mkP</code> over each input vector, but with possibly fewer <code>mkP</code> resources.</td></tr>
<tr><td></td><td><pre>function Pipe #(Vector #(mk, a),
                Vector #(mk, b))

         mkMap_with_funnel_indexed
               (UInt #(m) dummy_m,
                Pipe #(Tuple2 #(a, UInt #(logmk)), b) mkP,
                Bool param_buf_unfunnel)

        provisos (...);</pre></td></tr>
<tr><td></td><td><pre>module mkMap_with_funnel_indexed
                #(UInt #(m) dummy_m,
                  Pipe #(Tuple2 #(a, UInt #(logmk)), b) mkP,
                  Bool param_buf_unfunnel,
                  PipeOut #(Vector #(mk, a)) po_in)
                (PipeOut #(Vector #(mk, b)))

        provisos (...);</pre></td></tr>
</table>

This function captures a common pipeline design pattern. This is illustrated in Fig. 11. It has the

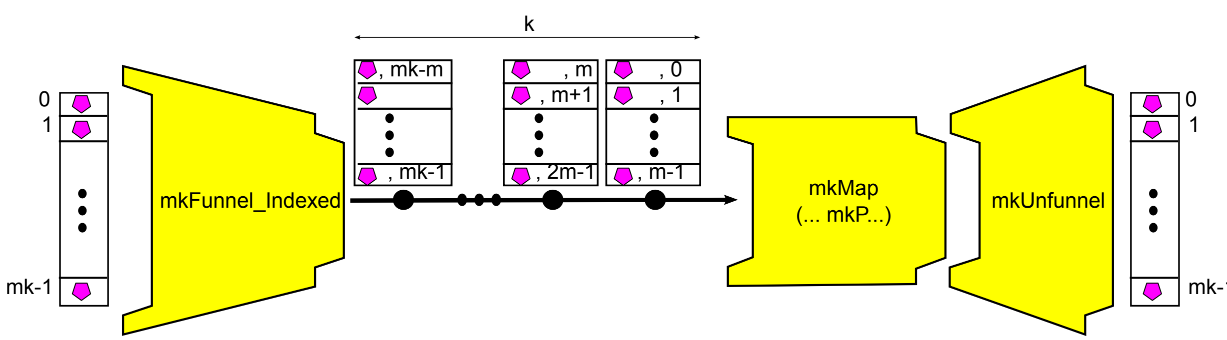

Figure 11: Funneling and unfunneling to constrain `mkP` resources

same logical functionality as `mkMap`, but instead of using `mk` copies of the mapped function `mkP`, uses only `m` copies, where `m` is a parameter. It is basically a composition of `mkFunnel_indexed`, followed by a narrower `mkMap` followed by a `mkUnfunnel`.

The input vector is first sliced into as `k`-sequence of smaller vectors (width `m`) where each element is tupled with its original index. That is, the input to the mapped function is of type

Tuple2 #(a, UInt #(logmk)). The elements of each slice are fed in parallel through m copies of mkP. The m results (of type b) form a slice of the output vector. These slices are unfunneled back into the output vector.

As in mkMap, the pipe mkP can have any latency, even data-dependent latency, and in fact the different instances of mkP do not have to have equal latency.

The parameter dummy_m specifies the desired m (degree of funneling). The parametric information is completely carried in the *type* of the parameter, so the actual value of the parameter is irrelevant (hence the name dummy_m). The parameter mkP represents the mapped function. Finally, the boolean parameter param_buf_unfunnel is only relevant when m=mk, and controls, in this corner case, whether the unfunneling component should be buffered or not.

mkMap_fn_with_funnel_indexed

Create a Pipe module that maps a given combinational function fn over each input vector, but with a limit on the fn resources.

```
function Pipe #(Vector #(mk, a),
                Vector #(mk, b))

         mkMap_fn_with_funnel_indexed
               (UInt #(m) dummy_m,
                function b fn (Tuple2 #(a, UInt #(logmk)) xj),
                Bool param_buf_unfunnel)

        provisos (...);
```

```
module mkMap_fn_with_funnel_indexed
                #(UInt #(m) dummy_m,
                  function b fn (Tuple2 #(a, UInt #(logmk)) xj),
                  Bool param_buf_unfunnel,
                  PipeOut #(Vector #(mk, a)) po_in)
                (PipeOut #(Vector #(mk, b)))

        provisos (...);
```

This is just a slightly optimized version of mkMap_with_funnel_indexed where the mapping function is a combinational function and not a pipe of arbitrary latency. The tighter assumptions on latency permit a leaner implementation. The remaining parameters dummy_m and param_buf_unfunnel are the same as before.

A.8 Linear composition

These functions correspond to mathematical composition of functions, that is, a linear chaining of pipes.

`mkCompose`	Create a `Pipe` module that composes two given `Pipe` modules `mkP` and `mkQ`.	
	`function Pipe #(a, c) mkCompose (Pipe #(a, b) mkP,` `                                 Pipe #(b, c) mkQ);`	
	`module mkCompose` `                #(Pipe #(a, b) pab,` `                  Pipe #(b, c) pbc,` `                  PipeOut #(a) pa)` `                (PipeOut #(c));`	

The input of the module (of type `a`) is fed into `mkP`, the output of which (of type `b`) is fed into `mkQ`, the output of which (of type `c`) is the final output. This is illustrated in Fig. 12.

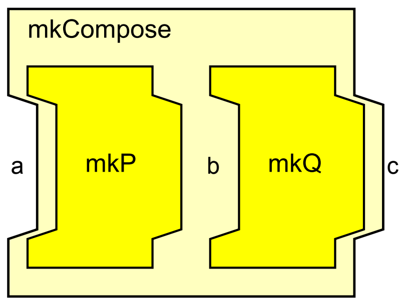

Figure 12: `mkCompose` cascades two pipes.

`mkCompose_buffered`	Create a `Pipe` module that composes two given `Pipe` modules `mkP` and `mkQ`, with an optional intermediate buffer.	
	`function Pipe #(a, c) mkCompose_buffered (Bool param_with_buffer,` `                                 Pipe #(a, b) mkP,` `                                 Pipe #(b, c) mkQ);`	
	`module mkCompose_buffered` `                #(Bool param_with_buffer,` `                  Pipe #(a, b) pab,` `                  Pipe #(b, c) pbc,` `                  PipeOut #(a) pa)` `                (PipeOut #(c));`	

This is functionally the same as `mkCompose`, except that the boolean parameter controls whether or not a `mkBuffer` is inserted between the two pipes (for data of type `b`).

`mkLinearPipe_S`	Create a `Pipe` module by repeatedly composing a `Pipe` module `mkStage` with itself `n` times.

```
function  Pipe #(a, a)
          mkLinearPipe_S
              (Integer n,
               function Pipe #(a,a) mkStage (UInt #(logn) j));
```

```
module mkLinearPipe_S
                #(Integer n,
                  function Pipe #(a,a) mkStage (UInt #(logn) j),
                  PipeOut #(a) po_in)
                (PipeOut #(a));
```

This is functionally similar to using `mkCompose` to cascade `n` copies of `mkStage`. Since the data output of one copy of `mkStage` is fed into the data input of another, its output and input types must be the same (type `a`). Here, `n` is a static parameter (hence the _S suffix). Each instance of `mkStage` is applied (statically) to its index `j` (0..`n`-1), so each `mkStage` can perform a different transformation. This is illustrated in Fig. 13.

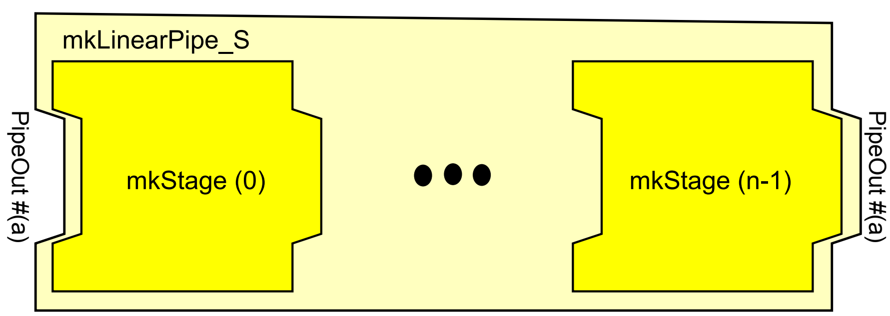

Figure 13: `mkLinearPipe_S` cascades n pipes.

`mkLinearPipe_S_Alt`	Alternative to `mkLinearPipe_S` with the same arguments and interface as `mkForLoop`, so that they can be substituted easily.

```
function  Pipe #(a, b)
          mkLinearPipe_S_Alt
              (Integer n,
              Pipe #(Tuple2 #(a, UInt #(wj)),
                     Tuple2 #(a, UInt #(wj)))  mkStage,
               Pipe #(a,b)  mkFinal);
```

```
module mkLinearPipe_S_Alt
                #(Integer   n,
                  Pipe #(Tuple2 #(a, UInt #(wj)),
                         Tuple2 #(a, UInt #(wj)))  mkStage,
                  Pipe #(a,b)  mkFinal,
                  PipeOut #(a)   po_in)
                (PipeOut #(b))
```

This creates a linear composition of n copies of `mkStage` followed by `mkFinal` (similar to a For-loop). The input to `mkStage` is a 2-tuple of a value and an incoming index. The stage normally passes the index through unchanged, but this is not a requirement. It may use the index internally for computation that depends on the index. The index emerging from a stage is incremented by `mkLinearPipe_S_Alt` before passing it into the next stage. The value emerging from the n^{th} stage is fed into `mkFinal`, and its output is the final output.

A.9 Forks and Joins

Forks and joins are illustrated in Fig. 14.

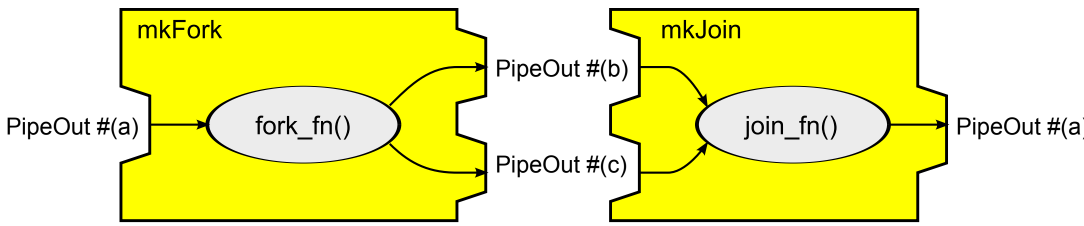

Figure 14: `mkFork` and `mkJoin`.

`mkFork`	Create a module that forks the output of one `PipeOut` interface into two other `PipeOut` interfaces, using a function `fork_fn` to split the input value.
	`module mkFork` `        #(function Tuple2 #(b, c) fork_fn (a va),` `          PipeOut #(a) poa)` `        (Tuple2 #(PipeOut #(b), PipeOut #(c)));`

This module receives an input value (of type `a`) from the input `PipeOut` interface `poa`; applies `fork_fn()` to this value to get a 2-tuple of values (of types `b` and `c` respectively); and feeds the first component of this tuple into its first `PipeOut` sub-interface and feeds the second component of the tuple into its second `PipeOut` sub-interface.

A typical `fork_fn` would simply replicate its input into both the outputs; another typical `fork_fn` splits its input into two components to be sent to the two outputs.

`mkForkVector`	Create a module that forks the output of one `PipeOut` interface into a vector of other `PipeOut` interfaces by replicating the input value.
	`module mkForkVector` `        #(PipeOut #(a) poa)` `        (Vector #(n, PipeOut #(a)));`

This module receives an input value (of type `a`) from the input `PipeOut` interface `poa`, replicates it, and sends each copy to one of the output vector of `PipeOut` interfaces.

`mkExplodeVector`	Create a module that receives a vector of values on one `PipeOut` interface and sends the j'th element into the j'th of a vector of `PipeOut` interfaces.
	`module mkExplodeVector #(PipeOut #(Vector #(n, a)) poa) (Vector #(n, PipeOut #(a)));`

This module receives a vector input values (of type `Vector #(a)`) from the input `PipeOut` interface `poa`, and sends the j^{th} element into the j^{th} `PipeOut` interface in the output vector of `PipeOut` interfaces.

`mkForkAndBufferRight`	Create a module that receives a value on one `PipeOut` interface and sends a copy to each of two `PipeOut` interfaces, buffering one of them.
	`module mkForkAndBufferRight #(PipeOut #(a) poa) (Tuple2 #(PipeOut #(a), PipeOut #(a)));`

This module receives an input value (of type `a`) from the input `PipeOut` interface `poa`, and sends a copy to both its output `PipeOut` interfaces. The second element of the tuple has an extra layer of buffering. This is a common structure in pipelines where a value both has to be used in the current stage (first element of tuple) and must also be forwarded to a later stage (second element of tuple).

`mkJoin`	Create a module that joins the output of one `PipeOut` interface into two other `PipeOut` interfaces, using a function `join_fn` to combine the input values.
	`module mkJoin #(function c join_fn (b vb, c vc), PipeOut #(b) pob, PipeOut #(c) poc) (PipeOut #(a));`

This module receives an input value (of type `b`) from the input `PipeOut` interface `pob` and an input value (of type `c`) from the input `PipeOut` interface `poc`. It combines these using `join_fn` into a value of type `a` and sends it into the output `PipeOut` interface.

A typical `join_fn` would simply tuple the two input values.

Note that this module effecticly "synchronizes" the outputs of the two incoming pipes, since it must wait for both inputs to be available before it can move any data.

A.10 Conditional Pipes (If-Then-Else Structures)

<table>
<tr><td rowspan="3"><code>mkIfThenElse</code></td><td>Create a <code>Pipe</code> module that conditionally executes either pipe <code>mkPT</code> or pipe <code>mkPF</code>.</td></tr>
<tr><td><pre>function Pipe #(Tuple2 #(a, Bool), b)
         mkIfThenElse (Integer latency,
                       Pipe #(a,b) mkPT,
                       Pipe #(a,b) mkPF)</pre></td></tr>
<tr><td><pre>module mkIfThenElse
                #(Integer latency,
                  Pipe #(a,b) pipeT,
                  Pipe #(a,b) pipeF,
                  PipeOut #(Tuple2 #(a, Bool)) poa)
                (PipeOut #(b));</pre></td></tr>
</table>

The input is a 2-tuple, of type `Tuple2 #(a,Bool)`. If the boolean is true, the value of type `a` is sent into pipe `mkPT`, otherwise it is sent into pipe `mkPF`. The results of `mkPT` and `mkPF` are the outputs of this module. This is illustrated in Fig. 15.

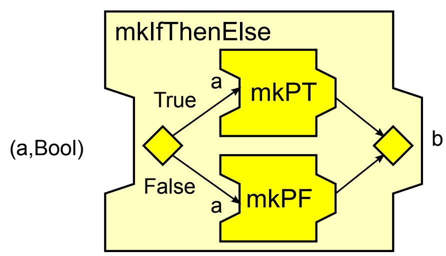

Figure 15: `mkIfThenElse` structure.

The pipes `mkPT` and `mkPF` can have different, even data-dependent latencies. Despite this, the internal control logic will ensure that I/O order is preserved. For example, a datum going through a long `mkPT` pipe cannot be "overtaken" to the output by a datum going through the alternative, shorter, `mkPF` path. Maintaining this ordering comes with a cost: internally, there is a FIFO of booleans that control the output ordering. the depth of this FIFO is controlled specifying the parameter `latency`. The circuit will work correctly even with this parameter specified as "1", but for full pipelined utilization of `mkPT` and `mkPF`, it should be set to the maximum latency of `mkPT` and `mkPF`.

If ordering does not matter in your application, use `mkIfThenElse_unordered`, described below, which does not incur this additional hardware cost.

Note this difference: a `mkFork` sends each datum down both paths; a `mkIfThenElse` sends it down only one of the two paths.

`mkIfThenElse_unordered`	Create a `Pipe` module that conditionally executes either pipe `mkPT` or pipe `mkPF`. Outputs may emege in a different order from the input order.
	`function Pipe #(Tuple2 #(a, Bool), b)` `        mkIfThenElse_unordered` `                    (Pipe #(a,b)  mkPT,` `                     Pipe #(a,b)  mkPF)`
	`module mkIfThenElse_unordered` `              #(Pipe #(a,b)  pipeT,` `                Pipe #(a,b)  pipeF,` `                PipeOut #(Tuple2 #(a, Bool)) poa)` `              (PipeOut #(b));`

This is a cheaper version of `mkIfThenElse`, where no attempt is made to preserve output ordering relative to inputs. For example, a datum going through a long `mkPT` pipe may be "overtaken" to the output by a datum going through the alternative, shorter, `mkPF` path. If ordering matters in your application, use `mkIfThenElse`.

Note this difference: a `mkFork` sends each datum down both paths; a `mkIfThenElse_unordered` sends it down only one of the two paths.

A.11 Looped Pipelines

The following describes while-loops (loop while condition is true) and for-loops (loop for n iterations). In both cases, the loops are truly pipelined, that is, multiple data items can circulate through the loop simultaneously. For example, suppose the loop body is a simple pipeline with 5 stages. Then, 5 data items can, like a railway train, circulate around the loop simultaneously. We call this the loop "capacity" c. The overall behavior will be: c items enter the loop; they circulate as many times around the loop as necessary; and then they depart, after which the next c items enter the loop, and so on. Of course, in a while-loop, they may not all circulate the same number of times, and so they may depart in a different order (and, as each one departs, a new item can be admitted).

Note, the loop capacity c may not be equal to the latency around the loop—it could be less. For example, if the loop body does sequential (non-pipelined) operations on each datum, it could increase the around-the-loop latency without increasing loop capacity. In any case, the loops below have control logic that will automatically allow sufficient data into the loop to reach its natural capacity.

`mkWhileLoop`	Create a `Pipe` module representing a while-loop from `Pipe` components representing the pre-test loop body, the post-test loop body, and the post-test loop postlude.
	<pre>function Pipe #(a, c) mkWhileLoop #(Pipe #(a,Tuple2 #(b, Bool)) mkPreTest, Pipe #(b,a) mkPostTest, Pipe #(b,c) mkFinal);</pre>
	<pre>module mkWhileLoop #(Pipe #(a,Tuple2 #(b, Bool)) mkPreTest, Pipe #(b,a) mkPostTest, Pipe #(b,c) mkFinal, PipeOut #(a) po_in) (PipeOut #(c));</pre>

An input to the loop (of type `a`) is fed into the pipe `mkPreTest`, which computes a 2-tuple of values with types `b` and `Bool`. If the boolean is `True`, the value of type `b` is fed into the pipe `mkPostTest`, and its output (of type `a`) is looped back into `mkPreTest`. Thus, a value circulates in this loop while the boolean is `True`.

If the boolean coming out of `mkPreTest` is `False`, the corresponding value of type `b` is fed into the postlude pipe `mkFinal`, whose output (of type `c`) is the output of this module. This is illustrated in Fig. 16.

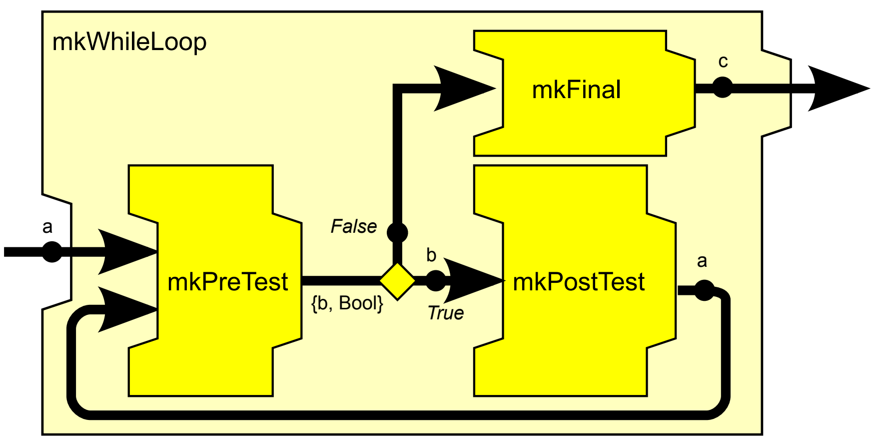

Figure 16: `mkWhileLoop` structure.

The pipes `mkPreTest`, `mkPostTest` and `mkFinal` can have any latency, including zero and data-dependent latency.

`mkWhileLoop` contains a `mkFIFOF` buffer at the head of the loop (holding a datum of type `a`). This is necessary because `mkPreTest` and `mkPostTest` need not have state—they could both be combinational, and in this case the FIFO serves to break the potential combinational loop.

<table>
<tr><td rowspan="3"><code>mkForLoop</code></td><td>Create a <code>Pipe</code> module representing a for-loop from <code>Pipe</code> components representing the loop body and the loop postlude.</td></tr>
<tr><td><pre>function Pipe #(a, b)
        mkForLoop
                (Integer   n_iters,
                 Pipe #(Tuple2 #(a, UInt #(wj)),
                        Tuple2 #(a, UInt #(wj))) mkLoopBody,
                 Pipe #(a,b)   mkFinal);</pre></td></tr>
<tr><td><pre>module mkForLoop
               #(Integer   n_iters,
                 Pipe #(Tuple2 #(a, UInt #(wj)),
                        Tuple2 #(a, UInt #(wj))) mkLoopBody,
                 Pipe #(a,b)   mkFinal,
                 PipeOut #(a)   po_in)
               (PipeOut #(b));</pre></td></tr>
</table>

This builds a pipeline representing a for-loop where the loop body `mkLoopBody` is iterated with indexes $j = 0, 1, \ldots$ `n_iters-1`. In each iteration, the current datum is tupled with its loop index (of type `Uint#(wj)` and fed into the loop body which transforms the datum and yields a 2-tuple containing the loop-body output value and the index.

The final datum is fed into `mkFinal`, and its output is the result of this module. This is illustrated in Fig. 17.

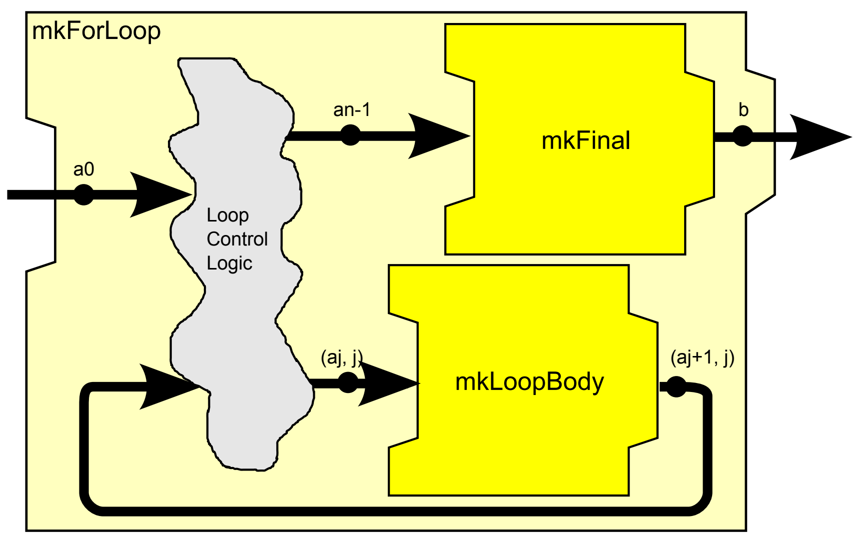

Figure 17: `mkForLoop` structure.

The pipes `mkLoopBody` and `mkFinal` can have any latency, including zero and data-dependent latency.

The loop index is passed through `mkLoopBody` for the following reasons: (1) The loop body may have conditionals dependent on the loop index. (2) For full pipeline utilization of `mkLoopBody`, the loop index should have the same latency through the loop body as the data computation itself, and this is best realized by the loop body itself. Many loop bodies do nothing but carry the index through unexamined and untouched. (3) The loop index can have a "stride" of $dj > 1$—the loop body can simply increment the loop index by $dj - 1$.

mkForLoop contains a mkFIFOF buffer at the head of the loop (holding a datum of type a). This is necessary because mkLoopBody need not have state—it could be combinational, and in this case the FIFO serves to break the potential combinational loop.

A.12 Reduction/Accumulating/Folding Pipelines

These are also looped pipelines, but instead of independently processing successive data, they combine them using a binary function to compute a result which is the output of the loop. For example, combining a sequence of numbers using the "+" function yields their sum. In mathematics this is also known as reduction, accumulation, or folding. This is illustrated in Fig. 18.

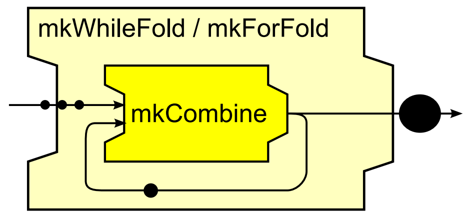

Figure 18: The "fold" constructors perform binary accumulations.

How many items should be combined before a result is yielded? The "while fold" accumulates while some condition holds, the "for folds" accumulate according to a given count.

mkWhileFold	Create a Pipe module from the Pipe component mkCombine that accumulates until a boolean signal.
	`function Pipe #(Tuple2 #(a, Bool), a) mkWhileFold (Pipe #(Tuple2 #(a,a), a) mkCombine);`
	`module mkWhileFold #(Pipe #(Tuple2 #(a,a), a) mkCombine, PipeOut #(Tuple2 #(a, Bool)) po_in) (PipeOut #(a));`

The inputs to this module are 2-tuples containing a value of type a and a boolean. The boolean represents an "last element of sequence" sentinel. As long as the boolean is False, the value is combined with the accumulation so far, using the pipe mkCombine which represents the binary combining function. When the boolean is True, that value is accumulated and the accumulated result is yielded as the result of this module.

The pipe mkCombine can have any latency, including zero, and data-dependent latency. The input sequences, the ends of which are marked by True, can be of any length, including 1. After one sequence, the next sequence can begin on the very next cycle.

mkWhileFold contains a mkPipelineFIFOF internally to hold the partially-accumulated value (of type a).

mkForFold	Create a Pipe module from the Pipe component mkCombine that accumulates n_items.	
	`function Pipe #(a, a) mkForFold (UInt #(wj) n_items, Pipe #(Tuple2 #(a,a), a) mkCombine);`	
	`module mkForFold #(UInt #(wj) n_items, Pipe #(Tuple2 #(a,a), a) mkCombine, PipeOut #(a) po_in) (PipeOut #(a));`	

```
function Pipe #(a, a)
        mkForFold
                (UInt #(wj) n_items,
                 Pipe #(Tuple2 #(a,a), a)  mkCombine);
```

```
module mkForFold
                #(UInt #(wj) n_items,
                  Pipe #(Tuple2 #(a,a), a)  mkCombine,
                  PipeOut #(a) po_in)
                (PipeOut #(a));
```

This module accumulates `n_items` input items using the pipe `mkCombine` which represents the binary combining function. The accumulated result is yielded as the result.

The pipe `mkCombine` can have any latency, including zero, and data-dependent latency. `n_items` must be ≥ 1. After one sequence, the next sequence can begin on the very next cycle.

`mkForFold` contains a `mkPipelineFIFOF` internally to hold the partially-accumulated value (of type `a`).

mkTreeReduceFn	Create a Pipe module doing pipelined binary tree reduction of input vectors.

```
function Pipe #(Vector#(n,a), a)
        mkTreeReduceFn (function a reduce2 (a x, a y),
                        function a reduce1 (a x),
                        Bit#(32)  addBuffer);
```

```
module mkTreeReduceFn #(function a reduce2 (a x, a y),
                        function a reduce1 (a x),
                        Bit#(32)  addBuffer,
                        PipeOut#(Vector#(n,a)) pipein)
                       (PipeOut#(a));
```

This module receives a vector of `n` values of type `a` on its input `PipeOut` interface, performs a binary tree reduction of these values, and outputs the result on its output `PipeOut` interface. At each level of the tree, `reduce2` is applied to adjacent pairs of vector elements, and if there is an odd element left, `reduce1` is applied to it (thus, `n` need not be a power of 2). The parameter `addBuffer` specifies which levels of the binary tree must be buffered: `addBuffer[0]` is for the lowest (widest, closest to input) level of the tree, `addBuffer[1]` is for the next level up, and so on until the root (output). The parameter is sized at `Bit#(32)` since your tree is unlikely to be so large as to have more than 32 levels.

<table>
<tr><td rowspan="3"><code>mkTreeReducePipe</code></td><td>Create a <code>Pipe</code> module doing pipelined binary tree reduction of input vectors.</td></tr>
<tr><td><code>function Pipe #(Vector#(n,a), a)</code>
<code>   mkTreeReducePipe (Pipe#(Tuple2#(a,a), a) reducepipe,</code>
<code>   Bit#(32) addBuffer);</code></td></tr>
<tr><td><code>module mkTreeReducePipe #(Pipe#(Tuple2#(a,a), a) reducepipe,</code>
<code>   Bit#(32) addBuffer,</code>
<code>   PipeOut#(Vector#(n,a)) pipein)</code>
<code>   (PipeOut#(a) ifc);</code></td></tr>
</table>

This module receives a vector of `n` values of type `a` on its input `PipeOut` interface, performs a binary tree reduction of these values, and outputs the result on its output `PipeOut` interface. Unlike `mkTreeReduceFn` described earlier, whose argument is a combinational binary reduction function, here the argument `reducepipe` is itself a pipe, taking a 2-tuple input and producing and output. Unlike `mkTreeReduceFn`, here, `n` must be a power of 2 (you will get a compile-time error otherwise). Just like `mkTreeReduceFn`, the parameter `addBuffer` specifies which levels of the binary tree must be buffered: `addBuffer[0]` is for the lowest (widest, closest to input) level of the tree, `addBuffer[1]` is for the next level up, and so on until the root (output). The parameter is sized at `Bit#(32)` since your tree is unlikely to be so large as to have more than 32 levels.

A.13 Reordering

In some pipelines where different data items encounter different latencies through different paths, a data item can "overtake" another, so that ordering is not preserved. Two examples:

- In `mkIfThenElse_unordered` described in Section A.10, the two arms of the conditional may have different latencies and thereby allow overtaking.
- In `mkWhileLoop` described in Section A.11, different data items may circulate a different number of times, allowing overtaking.

The following module is an order-restoring wrapper around pipelines that may not preserve order, provide the component pipeline is willing to carry a "tag" of type `CBToken#(n)` through, acoompanying the actual data.

<table>
<tr><td rowspan="3"><code>mkReorder</code></td><td>Create a <code>Pipe</code> module with ordered I/O from the <code>Pipe</code> component <code>mkBody</code> that may not have ordered I/O.</td></tr>
<tr><td><code>function Pipe #(a, b)</code>
<code>   mkReorder</code>
<code>   (Pipe #(Tuple2 #(CBToken #(n), a),</code>
<code>   Tuple2 #(CBToken #(n), b)) mkBody);</code></td></tr>
<tr><td><code>module mkReorder</code>
<code>   #(Pipe #(Tuple2 #(CBToken #(n), a),</code>
<code>   Tuple2 #(CBToken #(n), b)) mkBody,</code>
<code>   PipeOut #(a) po_in)</code>
<code>   (PipeOut #(b));</code></td></tr>
</table>

This module makes use of the `CompletionBuffer` package in the standard BSV library.

The `Pipe` module `mkBody` basically represents a function from data of type `a` to data of type `b`. In addition, it carries a "tag" of type `CBToken#(n)` along with each datum from input to output. The module treats the tag as an opaque object, that is, it does not examine the tag or compute with it; it simply carries the tag along with the data. The pipe can return results out of order.

The `mkReorder` function then makes a pipelined module from data of type `a` to data of type `b` in which I/O order is preserved, that is, it implements all the "scoreboarding" control logic necessary to restore order. This is illustrated in Fig. 19.

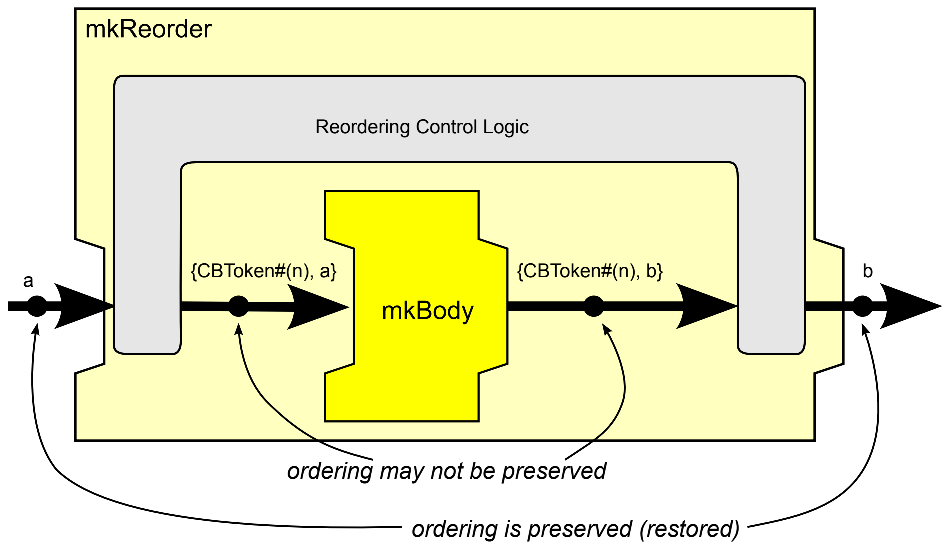

Figure 19: `mkReorder` restores ordering around out-of-order pipelines.

The pipe `mkBody` can have any latency, including zero, and data-dependent latency.

The parameter `n` specifies the desired "capacity" of the pipe, that is, the number of data items that can simultaneously be in flight through the `mkBody` module.

`mkReorder` contains, internally, an instance of a Completion Buffer of size `n`, which is roughly `n` buffers for data of type `b`.

B Prelude functions

The standard `Prelude` library is automatically inlcuded in all BSV packages. The package includes a set of functions which return functions which users of PAClib will find helpful.

<table>
<tr><td rowspan="2"><code>compose</code></td><td>Creates a new function, <code>c</code>, made up of functions, <code>f</code> and <code>g</code>. That is, <code>c(a) = f(g(a))</code></td></tr>
<tr><td><pre>function (function c_type (a_type x0))
     compose(function c_type f(b_type x1),
             function b_type g(a_type x2));</pre></td></tr>
</table>

`composeM`	Creates a new monadic function, `m#(c)`, made up of functions, `f` and `g`. That is, `c(a) = f(g(a))`
	<pre>function (function m#(c_type) (a_type x0)) composeM(function m#(c_type) f(b_type x1), function m#(b_type) g(a_type x2)) provisos # (Monad#(m));</pre>

`id`	Identity function, returns `x` when given `x`. This function is useful when the argument requires a function which doesn't do anything.
	`function a_type id(a_type x);`

`constFn`	Constant function, returns `x`.
	`function a_type constFn(a_type x, b_type y);`

`flip`	Flips the arguments `x` and `y`, returning a new function.
	<pre>function (function c_type new (b_type y, a_type x)) flip (function c_type old (a_type x, b_type y));</pre>

`curry`	This function converts a function on a pair (Tuple2) of arguments into a function which takes the arguments separately. The phrase `t0 f(t1 x, t2 y)` is the function returned by `curry`
	<pre>function (function t0 f(t1 x, t2 y)) curry (function t0 g(Tuple2#(t1, t2) x));</pre>

`uncurry`	This function does the reverse of `curry`; it converts a function of two arguments into a function which takes a single argument, a pair (Tuple2).
	<pre>function (function t0 g(Tuple2#(t1, t2) x)) uncurry (function t0 f(t1 x, t2 y));</pre>

Index