1 TODOS

1. Need to add area/time estimates for each element and application
2. How to keep input streams to map in sequence?
3. How does this work if I have a pipe operator that makes zipn more useful?

2 Type Notation

2.1 Combinational Element Types

A combinational element is one that is implemented using only combinational logic. The type signature of combinational elements is:

```
input 0 type -> ... -> input n-1 type -> output
type
```

Since these combinational elements process all inputs and output every clock cycle, they do not consume nor produce ready or valid signals. If they are contained within a larger, nested pipeline that has ready-valid signals, they do no affect those signals. These elements are always ready and produce valid data on every cycle that they receive valid data.

Each input or output has type T or T[p]. T is a base type, and it can contain nested types like arrays that are not relevant for the current operation. T[p] is an array of length p of T's. T[p][q] is an array of length p of arrays of length q of T's.

2.2 Sequential Element Types

A sequential element is one that has sequential logic in its implementation. The type signature of sequential elements is:

```
{input 0 num cycles, input 0 type per cycle} ->
... -> {input n-1 num cycles, input n-1 type
per cycle} ->
{ouput num cycles, output type per cycle}
```

One of the inputs or outputs of a stream may have different types during different clock cycles in a stream. For example, a sequential reduce may emit invalid data for most of the stream's cycles and then emit the result on the final cycle of the stream. Instead of {cycles, type for all cycles}, this type is represented in the following way:

```
{type at cycle 0, type at cycle 1, ..., type at
cycle n-1}
```

For short-hand where a type is the same for multiple cycles:

```
{type at cycle 0(0:n-2), type at cycle n-1}
```

The invalid type is represented as $\emptyset$.

These elements interfaces also must have clock inputs and may have ready-valid inputs and outputs. These ready-valid handshake ports indicate:

1. ready

   input: indicates to this sequential element that the next one in the pipeline has completed its prior input stream and is ready to receive more input.

   output: indicates to the previous sequential element in the pipeline that this one has completed its prior input stream and is ready to receive more input.

2. valid

   input: indicates to this sequential element that the previous one in the pipeline is emitting valid data that this sequential element can use as input.

   output: indicates to the next sequential element in the pipeline that this one is emitting valid data that the next one can use as input.

3 Elements

3.1 Basic Combinational Elements

These are elements that are built using logical primitives not available in Aetherling elements.

1. `tuple :: S -> T -> (S,T)`
2. `lb p w :: T[p] -> T[w+p-1]`
3. `overlap\_partition p w :: T[w+p-1] -> T[p][w]`
4. `partition p k :: T[k] -> T[k/p][p]`
5. `flatten p k :: T[k/p][p] -> T[k]`
6. `map p f ::` S_0`[p] -> ... ->` S_{m-1}`[p] -> T[p]`

   s.t. f :: S_0 -¿ ... -¿ S_{m-1} -¿ T
7. reduce p f :: T[p] -¿ T

   s.t. f :: (T,T)-¿ T
8. up k :: T -¿ T[k]
9. down k :: T[k] -¿ T
10. zip :: (S[k],T[k]) -¿ (S,T)[k]
11. unzip :: (S,T)[k] -¿ (S[k], T[k])
12. mem_read p :: ∅ -¿ T[p]
13. mem_write p :: T[p] -¿ ∅

3.2 Sequential Elements

3.2.1 Rate Changing Elements

1. serialize p :: {T[p], ∅(1:p-1)} -¿ {p, T}
2. deserialize p :: {p, T} -¿ {∅(0:p-2), T[p]}

3.2.2 Basic Sequential Versions Of Basic Combinational Elements

These are the sequential elements that cannot be implemented using only the basic combinational elements. In an ideal world, there would only be basic combinational logic blocks and rate changing sequential elements, but these elements are better implemented using logical blocks that do not belong in Aetherling.

1. reduce_seq k f :: {T[k], ∅(1:k-1)} -¿ {∅(0:k-2), T}

   s.t. f :: (T,T)-¿ T

2. reduce_seq_stream k f :: {k, T} -¿ {∅(0:k-2), T}

   s.t. f :: (T,T)-¿ T

3. up_seq k :: {T, ∅(1:k-1)} -¿ {k, T}

4. down_seq k :: {k, T} -¿ {T, ∅(1:k-1)}

3.2.3 Composed Sequential Versions Of Basic Combinational Elements

These are the sequential elements that can be implemented using only the basic combinational elements.

1. map_seq k f :: $\{S_0[k], \emptyset(1:p-1)\}$ -¿ ... -¿ $\{S_{m-1}[p], \emptyset(1:p-1)\}$ -¿ {∅(0:p-2), T[p]}

   s.t. f :: $S_0 \rightarrow ... \rightarrow S_{m-1} \rightarrow T$

   This map takes all the inputs in on the first cycle of the stream and emits all the outputs on the final cycle of the stream.

   implementation: map_seq p f = deserialize p $ f $
   map m (serialize p)

   note that in the above implementation, the type for serialize contains all the different input types to map

2. map_seq_stream p f :: $\{S_0[p], \emptyset(1:p-1)\}$ -¿ ... -¿ $\{S_{m-1}[p], \emptyset(1:p-1)\}$ -¿ {p, T}

   s.t. f :: $S_0 \rightarrow ... \rightarrow S_{m-1} \rightarrow T$

   This map takes all the inputs in on the first cycle of the stream and emits one output on each cycle of the stream.

   implementation: map_seq_stream p f = f $ map m (serialize p)

3. map_partially_par k p f :: $\{S_0[k], \emptyset(1:\frac{k}{p}-1)\}$ -¿ ... -¿ $\{S_{m-1}[k], \emptyset(1:\frac{k}{p}-1)\}$ -¿ $\{\emptyset(0:\frac{k}{p}-2), T[p]\}$

   s.t. f :: $S_0 \rightarrow ... \rightarrow S_{m-1} \rightarrow T$

implementation: map_partially_par k p f = flatten p k $
deserialize $\frac{k}{p}$ $ map p f $ map m (serialize $\frac{k}{p}$) $ map m (partition p k)

4. map_partially_par_stream k p f :: {S_0[k], $\emptyset(1{:}\frac{k}{p}-1)$} -¿ ... -¿ {$S_{m-1}$[k], $\emptyset(1{:}\frac{k}{p}-1)$} -¿ {$\frac{k}{p}$, T[p]}

   s.t. f :: $S_0 \rightarrow ... \rightarrow S_{m-1} \rightarrow$ T

   implementation: map_partially_par k p f = map p f $
   map m (serialize $\frac{k}{p}$) $ map m (partition p k)

5. reduce_partially_par k p f :: {T[p], $\emptyset(1{:}\frac{k}{p}-1)$} -¿ {$\emptyset(0{:}\frac{k}{p}-2)$, T}

   s.t. f :: (T,T) -¿ T

   implementation: reduce_seq k/p f $ map_seq k/p (reduce p f) $ partition p

6. reduce_partially_par_stream k p f :: {$\frac{k}{p}$, T[p]} -¿ {$\emptyset(0{:}\frac{k}{p}-2)$, T}

   s.t. f :: (T,T) -¿ T

   implementation: reduce_seq_stream k/p f $ reduce p f

4 Basic Applications

These are simple combinations of the basic elements.

4.1 Passthrough

1. mem_write 1 $ mem_read 1
2. mem_write t $ mem_read t

4.2 Array-Stream Conversions

1. mem_write 1 $ deserialize t $ mem_read t

   Note that mem_read fires onces every t'th clock cycle

2. mem_write t $ deserialize t $ serialize t $ mem_read t

   Note mem_write and mem_read fires every t'th clock cycle

3. mem_write 1 $ serialize t $ deserialize t $ mem_read 1

   Note this has a t cycle startup before mem_write starts writing

4.3 Map

1. mem_write 1 $ map 1 (+1) $ mem_read 1
2. mem_write t $ map t (+1) $ mem_read t
3. mem_write t $ map t (+1) $ map t f1 $ mem_read t
4. mem_write t $ map_seq t (+1) $ mem_read t

   Note mem_write and mem_read fire every t'th clock cycle
5. mem_write 1 $ map_seq_stream t (+1) $ mem_read t

   Note mem_read fires every t'th clock cycle
6. mem_write t $ map_partially_par t p (+1) $ mem_read t

   Note mem_write and mem_read fire every $\frac{t}{p}$'th clock cycle
7. mem_write 1 $ map_partially_par_stream t p (+1) $ mem_read t

   Note mem_read fires every $\frac{t}{p}$'th clock cycle

4.4 Reduce

1. mem_write 1 $ reduce t (+) $ mem_read t
2. mem_write 1 $ reduce t (+) $ deserialize t f $ mem_read 1

   Note everything after deserialize fires every t'th clock cycle
3. mem_write 1 $ reduce_seq t (+) $ mem_read t

   Note mem_write and mem_read fire every t'th clock cycle
4. mem_write 1 $ reduce_seq_stream t (+) $ mem_read 1
5. mem_write 1 $ reduce_partially_par t p (+) $ mem_read t

   Note mem_read fires every t'th clock cycle

6. mem_write 1 $ reduce_partially_par_stream t p (+) $ mem_read p

   Note mem_write fires every $\frac{t}{p}$'th clock cycle

4.5 Array Dimension Conversions

1. mem_write $\frac{t}{p}$ $ partition p t $ mem_read t

   Note that the element type mem_write is writing is T[p], and it writes $\frac{t}{p}$ of them every clock.

2. mem_write t $ flatten t $ partition t $ mem_read t
3. mem_write 1 $ flatten t $ partition t $ deserialize t $ mem_read t

   Note everything after deserialize fires onces every t'th clock cycle

4. mem_write 1 $ down t $ up t $ mem_read 1
5. mem_write 1 $ down_seq t $ up_seq t $ mem_read 1

5 Advanced Applications

5.1 Convolution

1. conv k p w