On The Use Of An Algebraic Language Interface For Waveform Definition

Michael L Dickens and J Nicholas Laneman

WIRELESS INSTITUTE
UNIVERSITY OF NOTRE DAME

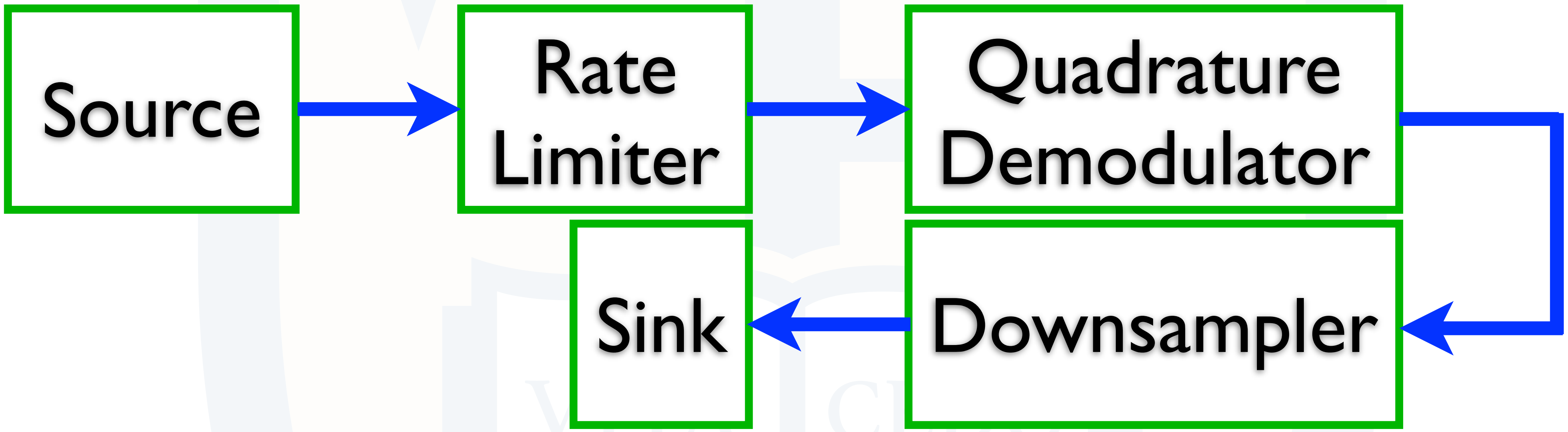
UNIVERSITY OF NOTRE DAME

WINNF'11'US 2011-Dec-02

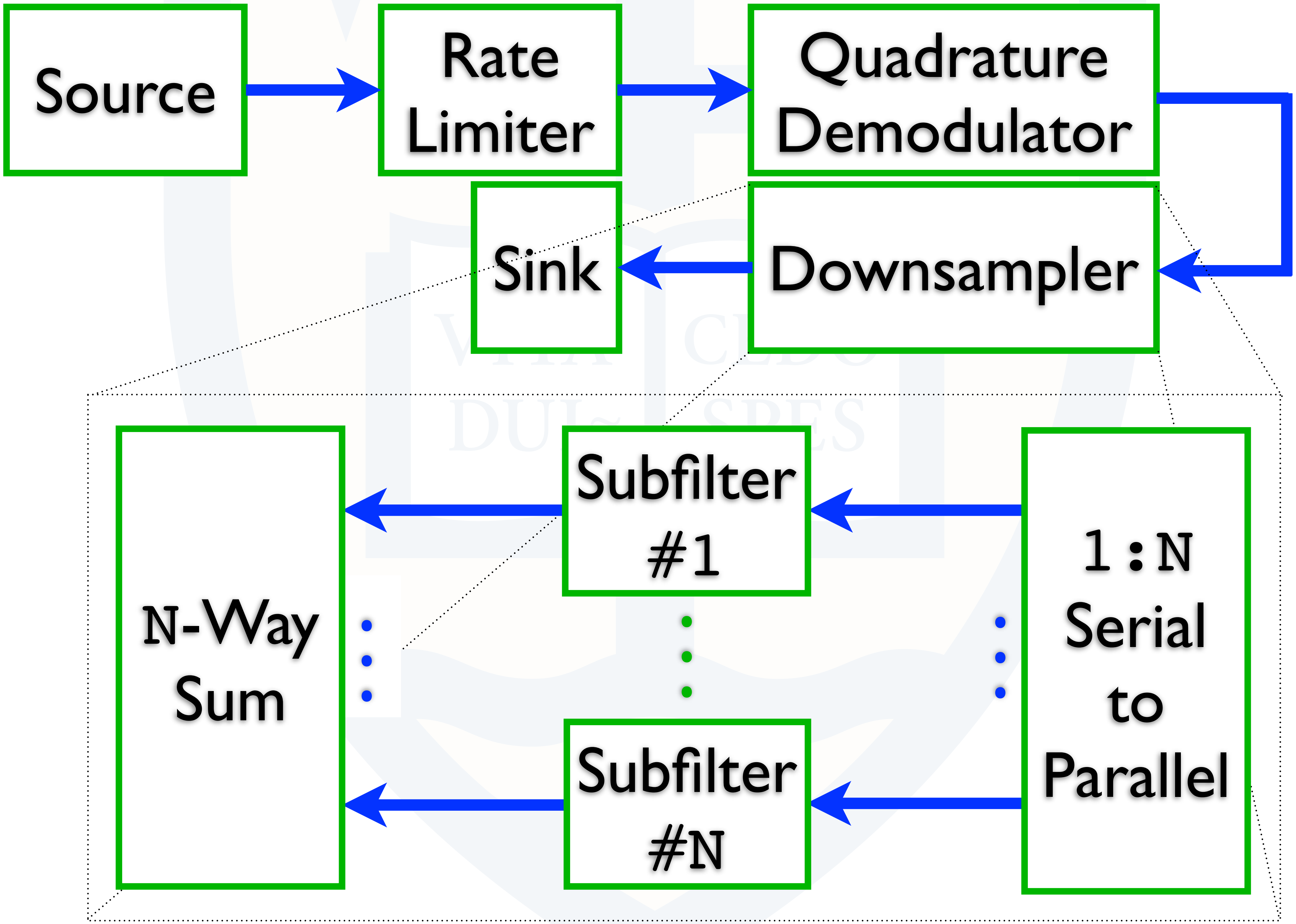
Overview

- Blocks versus Buffers
- Problem
- *Saline* Implementation
- Conclusions

A Waveform Graph



A Waveform Graph



Block-Centric Script

```
output = pp_down_N_block (input, N, options)
{
  s2p = serial_to_parallel (N, options)
  for n = 1:N {
   filter[n] = fir_filter (options.ppf[n])
  }
  acc = sum (options)
  connect ((input, 1), (s2p, 1))
  for n = 1:N {
    connect ((s2p, n), (filter[n], 1))
    connect ((filter[n], 1), (acc, n))
  }
  return (acc)
}
```

Block-Centric Script

```
output = pp_down_N_block (input, N, options)
{
  s2p = serial_to_parallel (N, options)
  for n = 1:N {
   filter[n] = fir_filter (options.ppf[n])
  }
  acc = sum (options)                              1
  connect ((input, 1), (s2p, 1))
  for n = 1:N {
    connect ((s2p, n), (filter[n], 1))
    connect ((filter[n], 1), (acc, n))
  }
  return (acc)
}
```

Block-Centric Script

```
output = pp_down_N_block (input, N, options)
{
  s2p = serial_to_parallel (N, options)
  for n = 1:N {
   filter[n] = fir_filter (options.ppf[n])
  }
  acc = sum (options)                          1
  connect ((input, 1), (s2p, 1))
  for n = 1:N {
    connect ((s2p, n), (filter[n], 1))
    connect ((filter[n], 1), (acc, n))
  }                                            2
  return (acc)
}
```

Buffer-Centric Script

```
output = pp_down_N_buffer (input, N, options)
{
  s2p = serial_to_parallel (input, N, options)
  acc = fir_filter (s2p[1], options.ppf[1])
  for n = 2:N {
    acc += fir_filter (s2p[n], options.ppf[n])
  }
  return (acc)
}
```

Buffer-Centric Script

```
output = pp_down_N_buffer (input, N, options)
{
  s2p = serial_to_parallel (input, N, options)
  acc = fir_filter (s2p[1], options.ppf[1])
  for n = 2:N {
    acc += fir_filter (s2p[n], options.ppf[n])
  }
  return (acc)
}
```

Buffer-Centric Script

```
output = pp_down_N_buffer (input, N, options)
{
  s2p = serial_to_parallel (input, N, options)
  acc = fir_filter (s2p[1], options.ppf[1])
  for n = 2:N {
    acc += fir_filter (s2p[n], options.ppf[n])
  }
  return (acc)
}
```

Buffer-Centric Script

```
output = pp_down_N_buffer (input, N, options)
{
  s2p = serial_to_parallel (input, N, options)
  acc = fir_filter (s2p[1], options.ppf[1])
  for n = 2:N {
    acc += fir_filter (s2p[n], options.ppf[n])
  }
  return (acc)
}
```

➡ Needs to be defined

- Means for defining functions taking stream buffers as arguments
- Means for defining functions returning an operation

Buffer-Centric Script

```
output = pp_down_N_buffer (input, N, options)
{
  s2p = serial_to_parallel (input, N, options)
  acc = fir_filter (s2p[1], options.ppf[1])
  for n = 2:N {
    acc += fir_filter (s2p[n], options.ppf[n])
  }
  return (acc)
}
```

Buffer-Centric Script

```
output = pp_down_N_buffer (input, N, options)
{
  s2p = serial_to_parallel (input, N, options)
  acc = fir_filter (s2p[1], options.ppf[1])
  for n = 2:N {
    acc += fir_filter (s2p[n], options.ppf[n])
  }
  return (acc)
}
```

➡ Needs to be defined

- Operations taking 1 or more streams as input
- Means for storing the output of an operation

Buffer-Centric Script

```
output = pp_down_N_buffer (input, N, options)
{
  s2p = serial_to_parallel (input, N, options)
  acc = fir_filter (s2p[1], options.ppf[1])
  for n = 2:N {
    acc += fir_filter (s2p[n], options.ppf[n])
  }
  return (acc)
}
```

Buffer-Centric Script

```
output = pp_down_N_buffer (input, N, options)
{
  s2p = serial_to_parallel (input, N, options)
  acc = fir_filter (s2p[1], options.ppf[1])
  for n = 2:N {
    acc += fir_filter (s2p[n], options.ppf[n])
  }
  return (acc)
}
```

➡ Needs to be defined

- Means for creating a temporary variable storing the output of a prior operation
- Means for appending a stream to the input stream list of an operation

Block Versus Buffer

Block

- Various forms in use since the late 1960's
- All former and current data-flow style processing
- Instantiation and connection can be in any order
- Non-algebraic language interface structure

Buffer

- Various forms in use since the early 1970's
- MATLAB has more than 1 million users worldwide
- Waveform must be created from source(s) to sink(s)
- Algebraic-like language interface structure

Problem

To allow script-based waveform definition using C++ and buffer-centric programming

Problem

To allow script-based waveform definition using C++ and buffer-centric programming

➡ Uses some special C++ sauce ...

- Namespaces
- Templates
- Operation Overloading
- **`typeid`**

Saline Implementation

Surfer Algebraic Language INterfacE

- Basic Classes
- Variable Types
- Operator Types
- Type Propagation
- Runtime Operation Checks

Saline Variable Types

➡ Requires 3 basic classes

1. A *base class*

```
namespace saline {
  template < typename item_t >
  class stream_base;
}
```

➡ All stream-oriented variable classes are derived from this base class, such that one can always downcast to a `saline::stream_base` of the appropriate type

Saline Variable Types

2. An *operator* class that represents the output buffer(s) resulting from some specific operator. For example, an **`fft`** operator class might be defined via

```
namespace saline {
  template < typename in_t,
             typename proc_t,
             typename out_t >
  class fft :
    public stream_base < out_t >;
}
```

➡ Only the output buffer type of the new class is provided to the base stream class

➡ Can be explicitly declared, but not required

Saline Variable Types

3. An *enclosure* variable class

```
namespace saline {
  template < typename item_t >
  class enclosure :
    public stream_base < item_t >;
}
```

- ➡ Contains a reference to an operator variable
- ➡ Can be explicitly declared
- ➡ Can be implicit temporary placeholders
  - e.g., when multiple operators are executed before the **`operator=`** method is issued
  - A new object is created and knowledge of this memory allocation is retained for later deletion

Saline Operator Types

- 6 primary operator types required to define an algebraic language

1. **`op (options)`**

   Operation taking no input streams, e.g., sources

2. **`op (stream1, …, streamN, options)`**

   Operation taken a-priori known number of input streams

3. **`op (stream1, …, options)`**

   Operation taken a number of input streams, which is not known until runtime

Saline Operator Types

4. **`stream1 op stream2 op stream3 …`**

   Generally expands at compile time to

   ```
   tmp = stream1 op stream2
   tmp op stream3
   ```

   where **`tmp`** is an implicit temporary enclosure variable. Expansion depends on language operator precedence ordering.

Saline Operator Types

4. **`stream1 op stream2 op stream3 …`**

   Generally expands at compile time to

   ```
   tmp = stream1 op stream2
   tmp op stream3
   ```

   where **`tmp`** is an implicit temporary enclosure variable. Expansion depends on language operator precedence ordering.

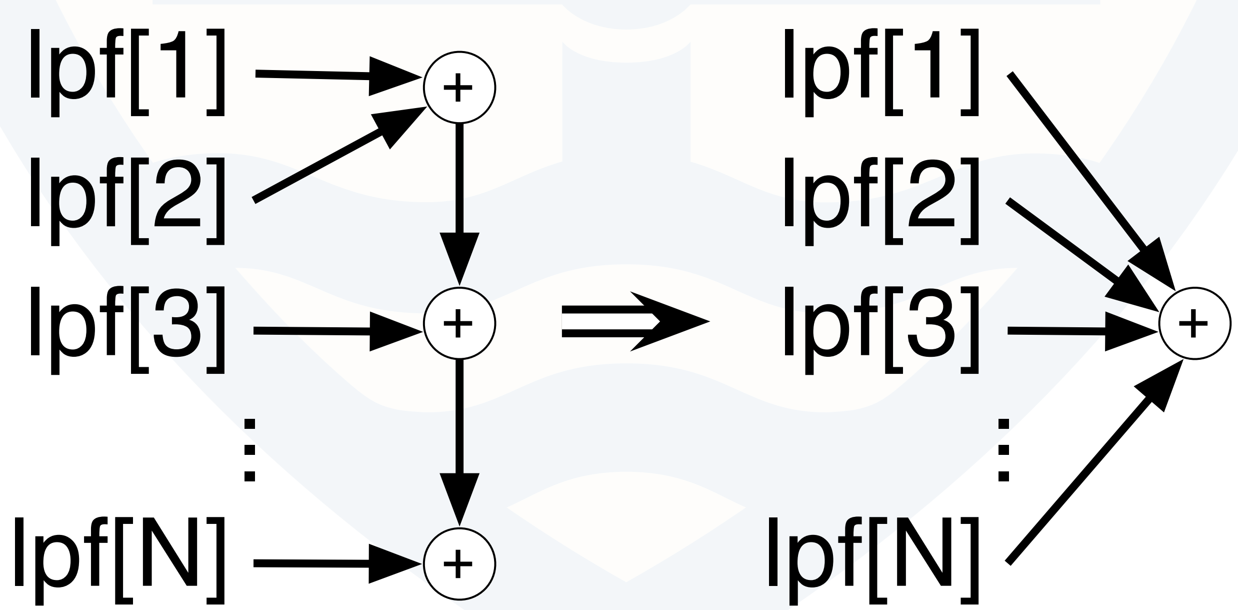
   Except ...

Saline Operator Types

4. **`stream1 op stream2 op stream3 …`**

… when all streams are of the same type, and all of the operators are the same, then runtime optimization can occur, e.g.,

`out = lpf[1] + lpf[2] + … + lpf[N]`



Saline Operator Types

5. **`stream1 = stream2`**

   Requires that **`stream1`** be an explicit enclosure variable. If **`stream2`** is an enclosure variable, then just copies the information held by **`stream2`** into **`stream1`**

6. **`stream1 op= stream2`**

   Requires that **`stream1`** be an explicit enclosure variable, and generally expands at runtime to

   ```
   tmp = stream1
   stream1 = tmp op stream2
   ```

   where **`tmp`** is an implicit temporary enclosure variable

Saline Operator Types

5. **`stream1 = stream2`**

   Requires that **`stream1`** be an explicit enclosure variable. If **`stream2`** is an enclosure variable, then just copies the information held by **`stream2`** into **`stream1`**

6. **`stream1 op= stream2`**

   Requires that **`stream1`** be an explicit enclosure variable, and generally expands at runtime to

   ```
   tmp = stream1
   stream1 = tmp op stream2
   ```

   where **`tmp`** is an implicit temporary enclosure variable

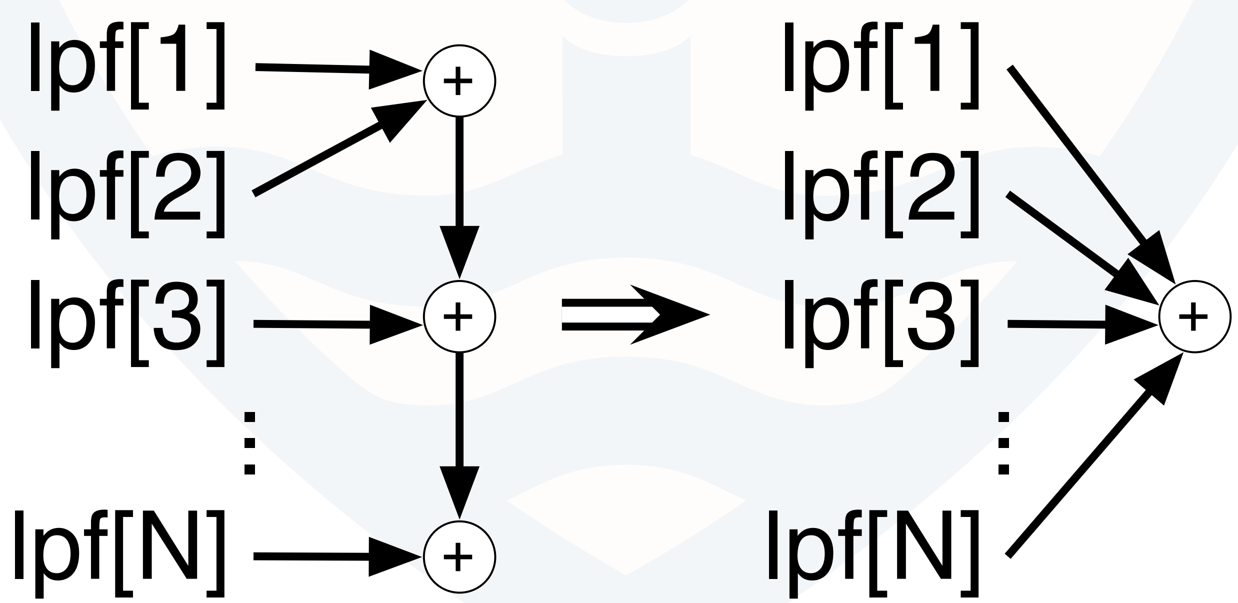
   Except ...

Saline Operator Types

6. **`stream1 op= stream2`**

when both streams are of the same type, and if **`stream2`** contains an operator of the same type as **`op`**, then runtime optimization can occur, e.g.,

```
out = lpf[1];
for n=2:N { out += lpf[n]; }
```



Type Propagation

- Operator types 1-4 return a **`saline::stream_base`** of some template type, e.g.

```
namespace saline {
  template < typename arg_t >
  stream_base < arg_t >&
  serial_to_parallel
  (stream_base < arg_t >& arg,
   int num_outputs,
   options_t& options);
}
```

- Stream type is propagated from input(s) to output(s) via the template parameter(s)

Runtime Operation Checks

- 3 checks are performed during runtime

1. Variable Overwriting : The code

```
saline::enclosure < int > A;
A = 5;
A = 10;
```

generates a warning on the last line, because the variable was overwritten. Internally, the last two lines of the above code are reinterpreted as

```
A = 5;
tmp_A = A;
A = 10;
```

where **tmp_A** is an implicit temporary enclosure variable

Runtime Operation Checks

2. Implicit type changes **:** The code

```
saline::enclosure < int > A;
saline::enclosure < float > B;
A = 5;
B = A;
```

generates a warning on the last line, because the stream type was not explicitly changed. Internally, the last two lines of the above code are reinterpreted as

```
tmp_A = saline::type_converter
            < int, float > (A);
B = tmp_A;
```

where **tmp_A** is an implicit temporary enclosure variable

Runtime Operation Checks

3. Variable declaration order **:** The code

```
saline::enclosure < int > A, B;
A = B;
```

generates an error on the last line, because the stream **B** has not been set before it is saved into stream **A**

Saline Code

```
namespace saline {
  template < typename arg_t >
  stream_base < arg_t > pp_down_N_Saline
  (stream_base < arg_t >& input,
   size_t N, options_t& options)
  {
    enclosure < arg_t > s2p, acc;
    s2p = serial_to_parallel (input, N, options);
    acc = fir_filter (s2p[1], options.ppf[1]);
    for (size_t n = 2; n < N; n++) {
      acc += fir_filter (s2p[n], options.ppf[n]);
    }
    return (acc);
  }
}
```

Conclusions

- Enabled algebraic-like waveform definition interface in C++
  - Buffer-centric approach to waveform definition
  - 3 variable type classes
  - 6 operator types, with possible runtime waveform optimization
  - 3 runtime operation checks
  - Stream type propagation via template arguments

Conclusions

➡ Enabled algebraic-like waveform definition interface in C++

- Buffer-centric approach to waveform definition
- 3 variable type classes
- 6 operator types, with possible runtime waveform optimization
- 3 runtime operation checks
- Stream type propagation via template arguments

Ongoing Work

➡ Increasing efficiency of runtime kernel

➡ More compelling example using OFDM

Thank you!

Questions?

Backup Slides

C++ Namespace

- Part of the C++ standard
- A namespace is the scope within which a given set of classes, functions, and global variables are valid
- Denoted by "::" between the namespace name (before), and the class, function, or variable (after), e.g.

```
namespace foo { int bar; }
```

- describes a variable **`bar`**, of type **`int`**, residing in the namespace **`foo`**. One could reference this variable directly after it is declared, via **`foo::bar`**
- Can have the same-named class, function, or variable in multiple namespaces, so there is a trade-off between too many and too few namespaces

C++ Templates

- Part of the C++ standard
- Allows a single definition to apply to any number of 'types'
- For example, the function **max** could be defined

```
template < typename T >
T max (T a, T b)
{ return (a > b ? a : b); }
```

- The above function could be used via, e.g.,

```
float fm = max < float > (1, 2);
```

- Recently ratified standard, C++11, allows for variable number of template arguments

C++ Operation Overloading

- Part of the C++ standard
- Define math operators, e.g., **+**, *****, **&**, **<**, **%**, for data-flows
- Overload the associated C++ operators, e.g., **operator+**, **operator***, etc..
- For example, **operator+** for identically-typed arguments

```
template < typename T > foo < T > operator+
(foo < T > lhs, foo < T > rhs) {
return (foo < T > (lhs.value () + rhs.value ())); }
```

- Using the above code, assuming **foo** is appropriately defined

```
foo < int > a, b, c;
a = 1;
b = 2;
c = a + b;
```

- Cannot do differently-typed arguments

C++ typeid

- Part of the C++ standard, but implementations vary from compiler to compiler
- Used for comparing any two already-declared variables' types
- For example, **operator+** for differently-typed arguments

```
template < typename lhs_t, typename rhs_t >
foo < lhs_t > operator+
(foo < lhs_t > lhs, foo < rhs_t > rhs) {
  lhs_t rhs_to_use = 0;
  if (typeid (lhs) == typeid (rhs)) {
    rhs_to_use = rhs.value ();
  } else {
    rhs_to_use = lhs_t (rhs.value ());
  }
  return (foo < lhs_t > (lhs.value () +
                         rhs_to_use));
}
```

C++ `typeid`

- Using the above code, assuming **`foo`** and **`operator=`** are appropriately defined

```
foo < int > a;
foo < short > b;
foo < long > c;
a = 1;
b = 2;
c = a + b;
```