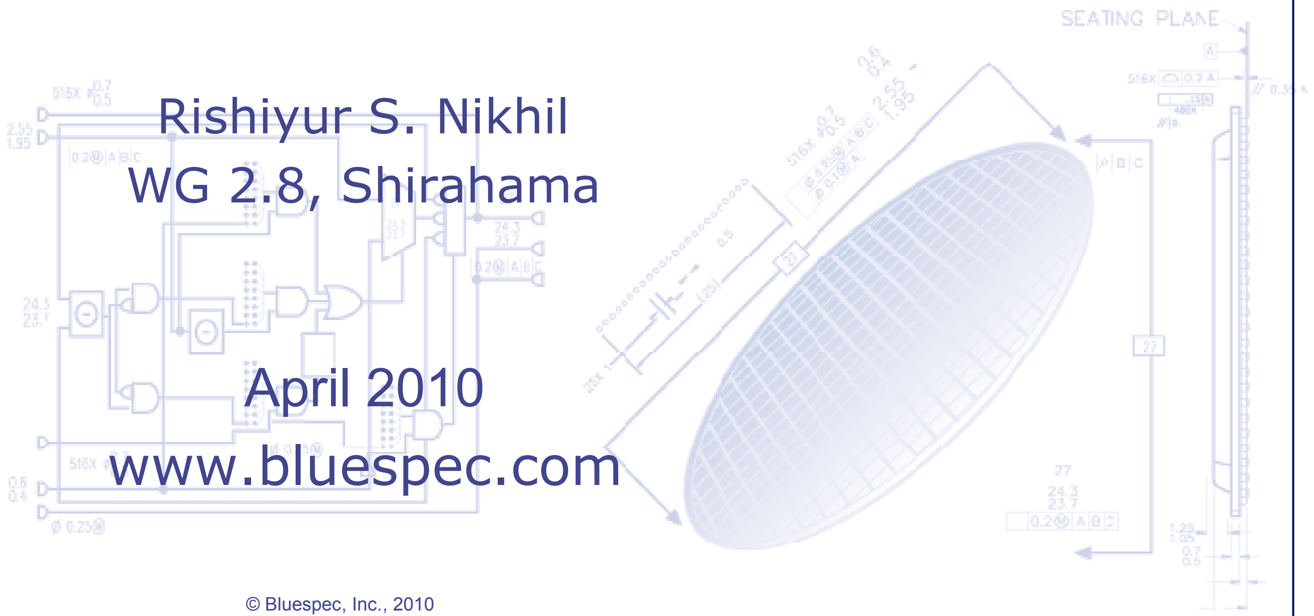
Two uses of FP techniques in HW design



Rishiyur S. Nikhil
WG 2.8, Shirahama

April 2010
www.bluespec.com

- PAClib: using HOFs for architectural flexibility
- HW Probes: using monads

Central importance of architecture

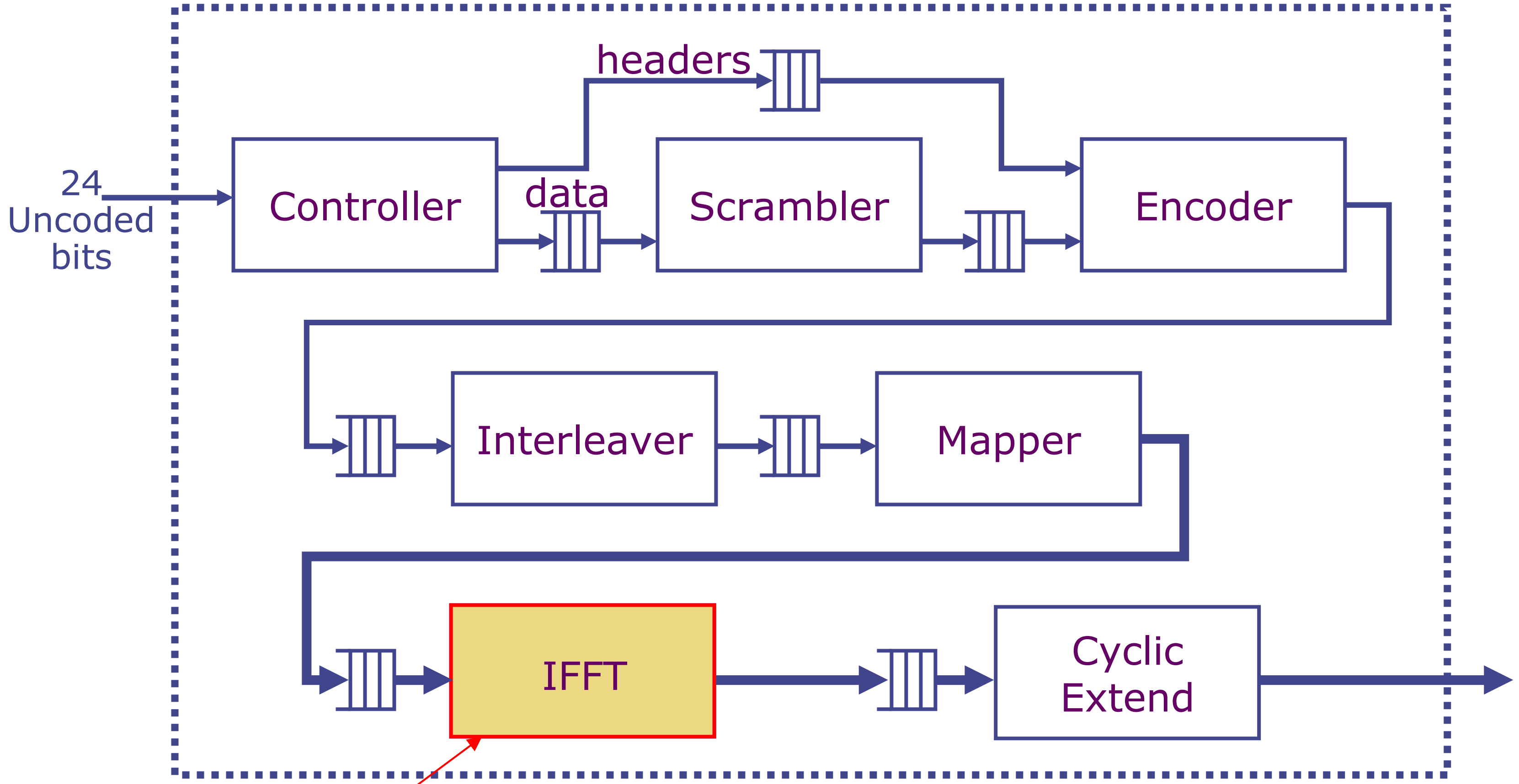
Metrics for HW design quality:

- Silicon area
- Latency/bandwidth (often translates to clock speed)
- Power consumption

Architecture is the first-order determinant of these metrics for HW

- (just like a good algorithm is, for SW)
- In HW design, architecture is inseparable from algorithm, because architecture determines the cost/computation model

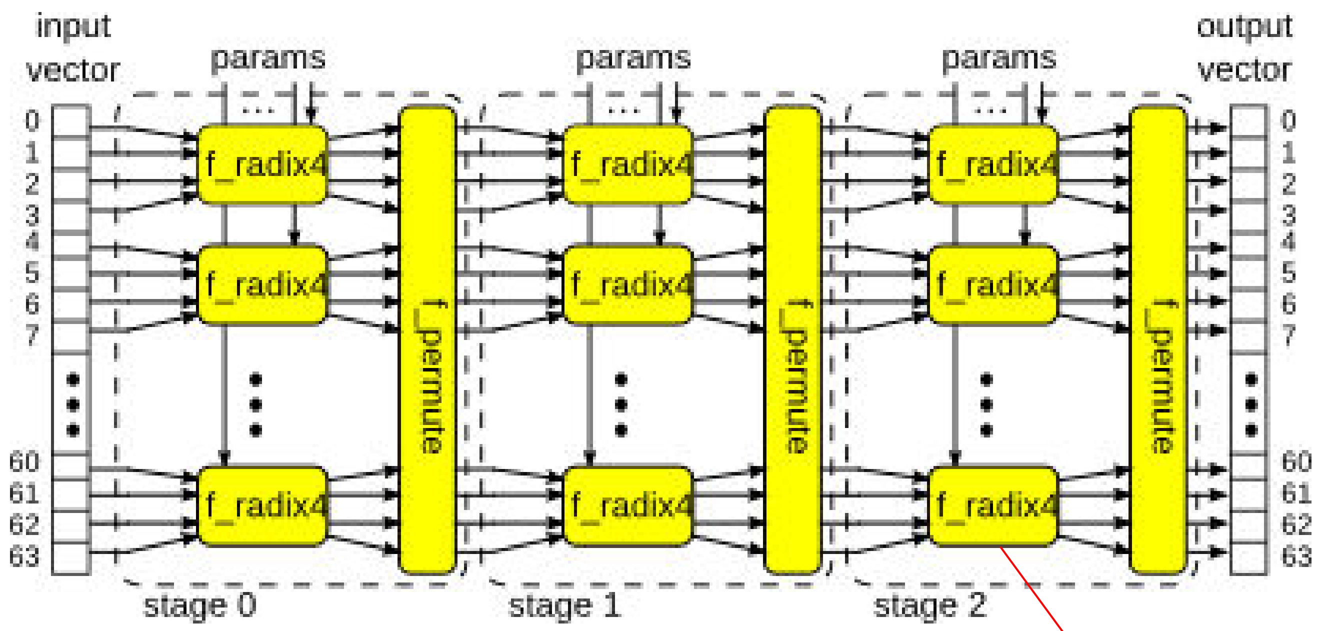
Example: IFFT in 802.11a (WiFi) transmitter

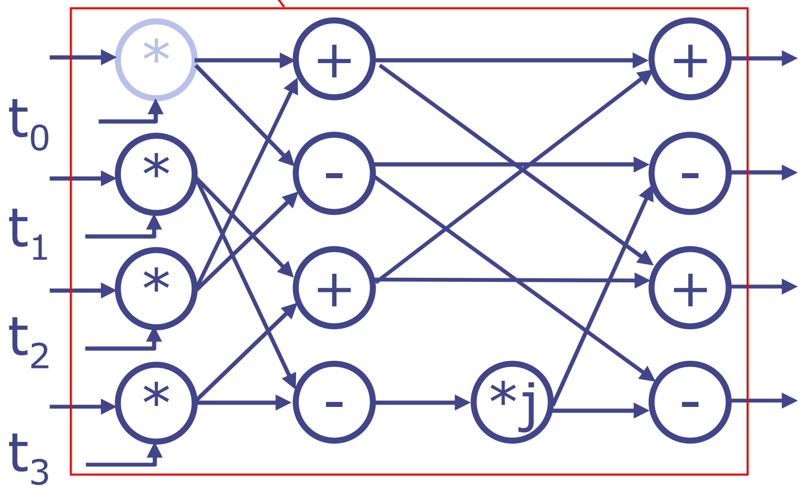


IFFT Transforms 64 (frequency domain) complex numbers into 64 (time domain) complex numbers

🟨 accounts for 85% area

The IFFT computation (math spec/data flow graph)

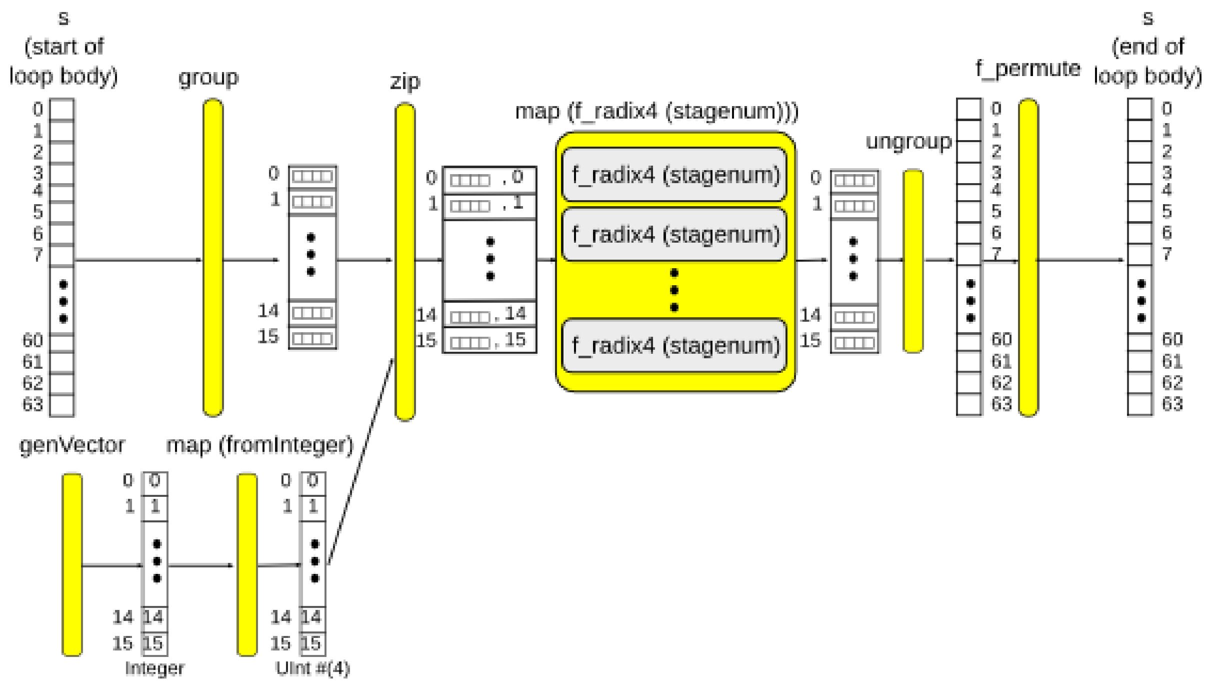




In this application of IFFT, all numbers are complex and represented as two sixteen bit quantities. Fixed-point arithmetic is used to reduce area, power, ...

For this discussion, f_radix4() and f_permute() are treated as black boxes (although they can also benefit from PAClib implementation)

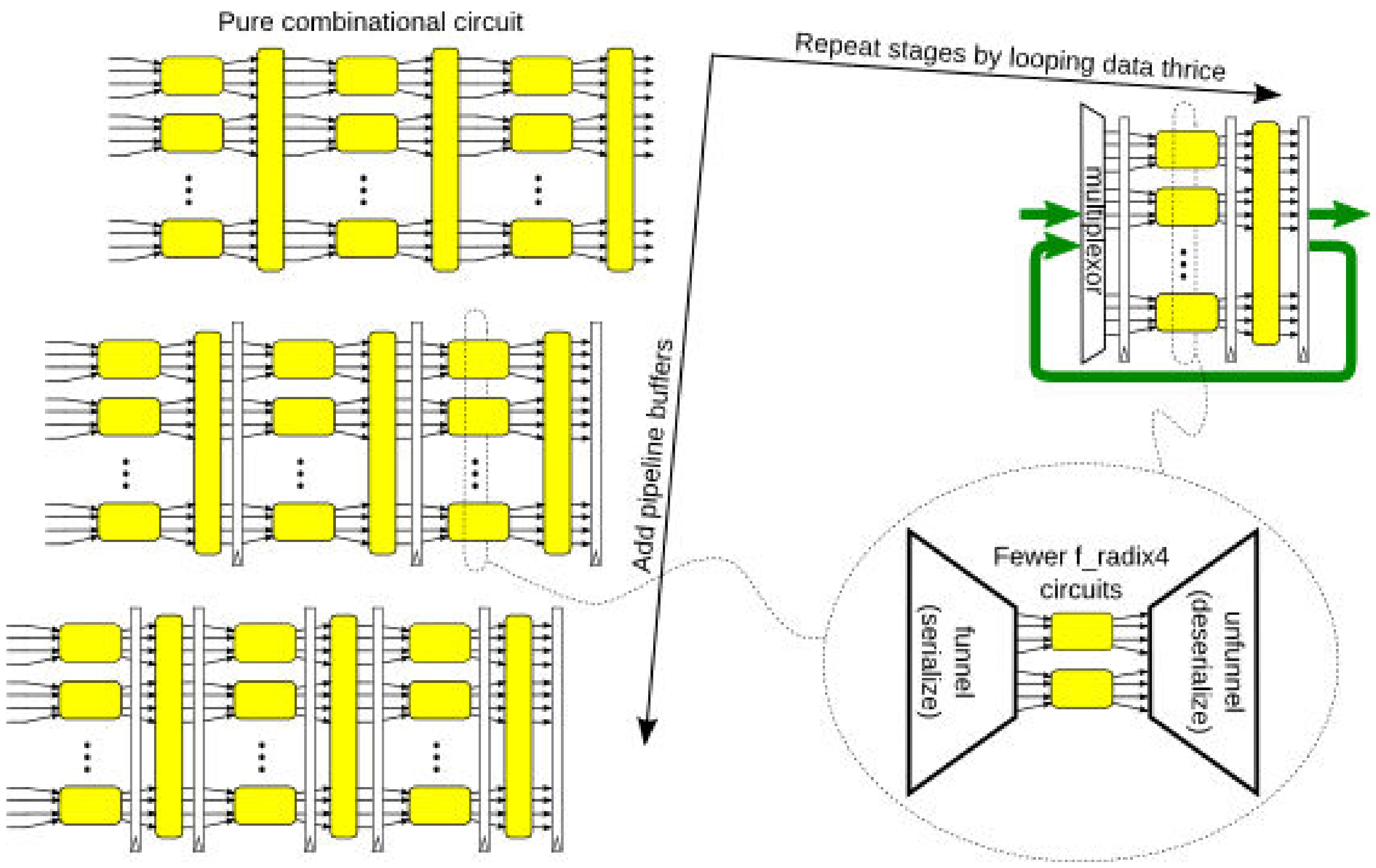
IFFT: BSV code for first functional model



```
function IFFTData f_IFFT (IFFTData s);
   for (UInt#(2) stagenum = 0; stagenum < 3; stagenum = stagenum + 1)
      s = f_permute (ungroup (map (f_radix4 (stagenum),
                                   zip (group (s),
                                        map (fromInteger, genVector)))));
   return s;
endfunction
```

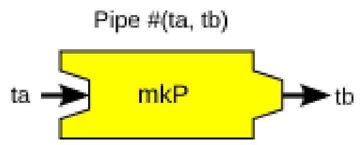
- *It's synthesizable (everything in BSV is synthesizable)*
- *FPGA emulation may be much faster than SW simulation*

IFFT: the HW architecture space



Basic Pipeline Interface

All pipeline components use a standard, parametrized (generic) interface:

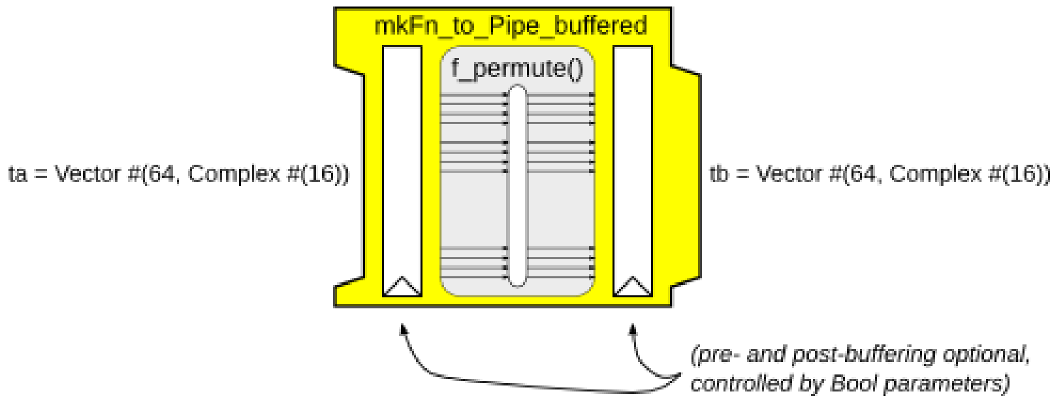


```
interface PipeOut #(type a);
   method a first ();
   method Action deq ();
   method Bool notEmpty ();
endinterface
```

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

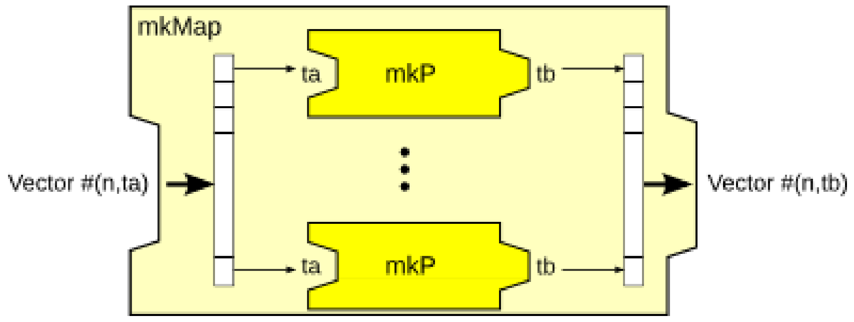
Wrapping a function into a Pipeline Component

```
function Pipe #(ta, tb) mkFn_to_Pipe_buffered (Bool paramPre,
                                                tb  fn (ta x),
                                                Bool paramPost);
```


mkFn_to_Pipe_buffered
f_permute()
ta = Vector #(64, Complex #(16))
tb = Vector #(64, Complex #(16))
(pre- and post-buffering optional, controlled by Bool parameters)

Build larger structures with higher-order functions

mkMap: given a pipe *mkP*, creates a pipeline component that sends each element of an input vector through a copy of *mkP*:



mkP, ta, tb, n are all parameters

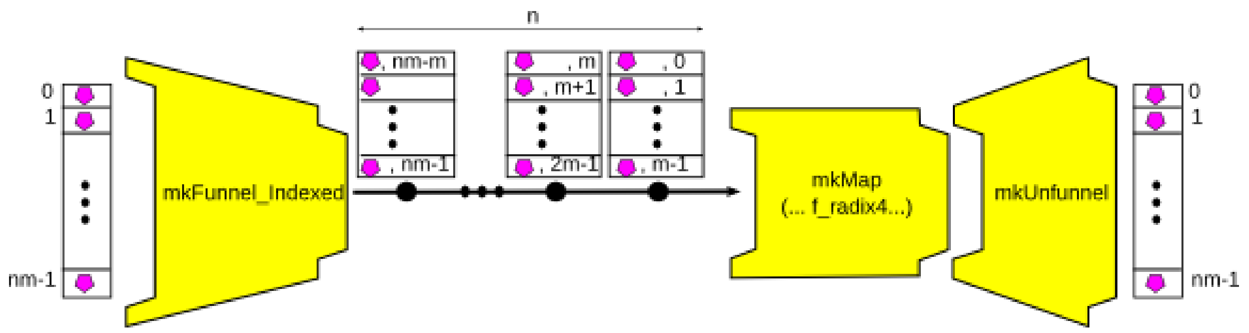
```
function Pipe #(Vector #(n, a),
                Vector #(n, b))
         mkMap (Pipe #(a, b) mkP);
```

Version of mkMap where each mkP is also given its index

```
function Pipe #(Vector #(n, a),
                Vector #(n, b))
         mkMap_indexed (Pipe #(Tuple2 #(a, UInt #(logn)), b) mkP);
```

Resource-constrained map

Resource-constrained maps: use *mkFunnel* and *mkUnfunnel* around *mkMap* to use fewer instances of *mkP*



n and *m* are parameters

```
function Pipe #(Vector #(nm, a),
                Vector #(m, Tuple2 #(a, UInt #(lognm))))
         mkFunnel_Indexed
         provisos (...);
```

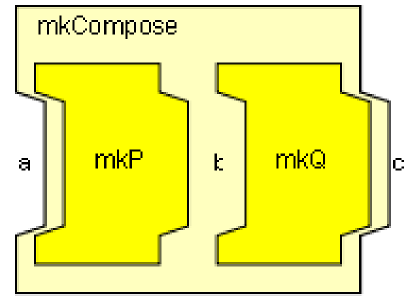
```
function Pipe #(Vector #(m, a),
                Vector #(nm, a))
         mkUnfunnel (Bool state_if_k_is_1)
         provisos (...);
```

```
function Pipe #(Vector #(nm, a),
                Vector #(nm, b))
         mkMap_with_funnel_indexed (UInt #(m) dummy_m,
                                    Pipe #(Tuple2 #(a, UInt #(lognm)), b) mkP,
                                    Bool param_buf_unfunnel)
         provisos (...);
```

Linear pipes

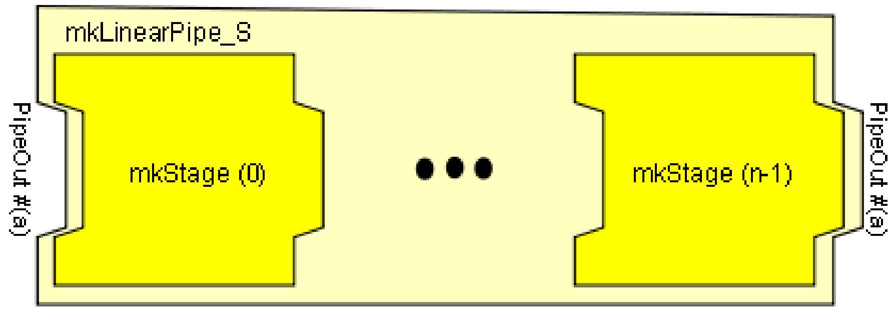
mkCompose

```
function Pipe #(a, c) mkCompose (Pipe #(a, b) mkP,
                                  Pipe #(b, c) mkQ);
```



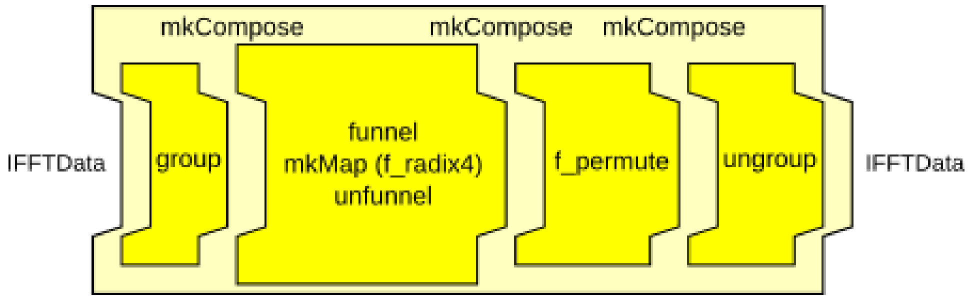
mkLinearPipe_S

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



Build larger structures with higher-order functions

mkStage_S

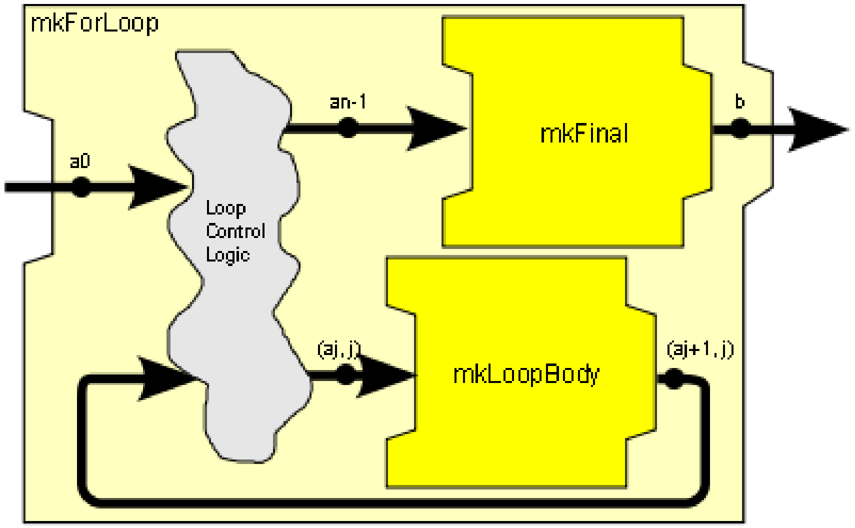


```
function Pipe #(IFFTData, IFFTData) mkStage_S (UInt#(2) stagenum);
   // ---- Group 64-vector into 16-vector of 4-vectors
   let grouper = mkFn_to_Pipe (group);

   // ---- Map f_radix4 over the 16-vec
   let mapper = mkMap_fn_with_funnel_indexed (param_dummy_m,
                                              f_radix4 (stagenum),
                                              param_buf_unfunnel);

   // ---- Ungroup 16-vector of 4-vectors into a 64-vector
   let ungrouper = mkFn_to_Pipe (ungroup);

   // ---- Permute it
   let permuter = mkFn_to_Pipe_buffered (False, f_permute,
                                         param_buf_permuter_output);

   return mkCompose (grouper,
                     mkCompose (mapper,
                                mkCompose (ungrouper, permuter)));
endfunction
```

Looped pipes

mkForLoop: compose stages:

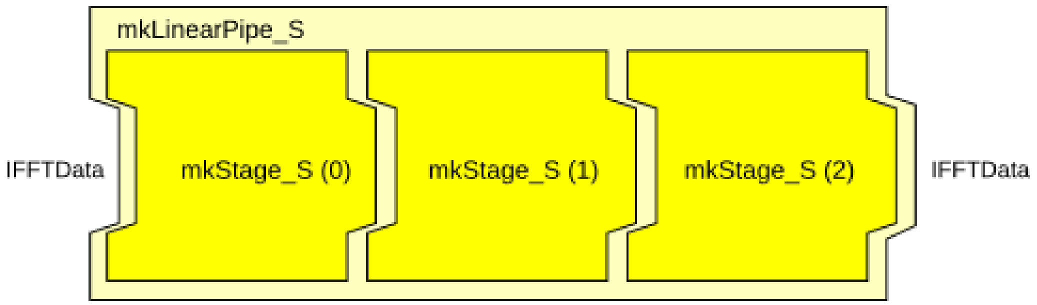
```
function Pipe #(a, b)
        mkForLoop (UInt #(wj)                              jmax,
                   Pipe #(Tuple2 #(a, UInt #(wj)),
                         Tuple2 #(a, UInt #(wj)))  mkLoopBody,
                   Pipe #(a,b)                     mkFinal);
```



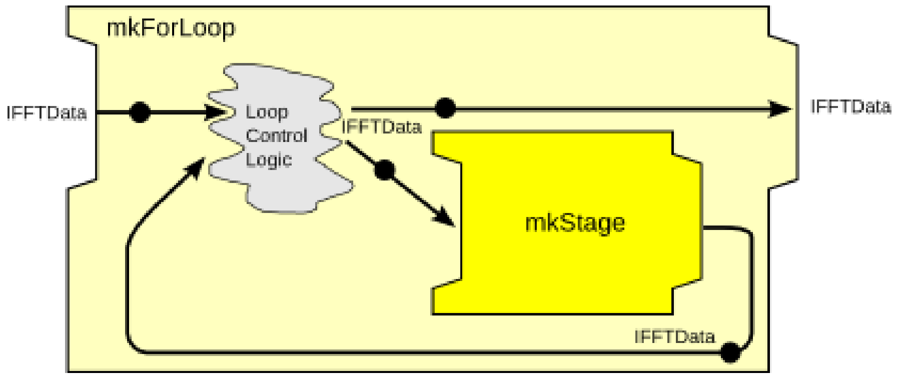
Note: there could be many samples simultaneously in the loop body

Top-level of IFFT

mkLinearPipe_S: compose stages:



mkForLoop: compose stages:



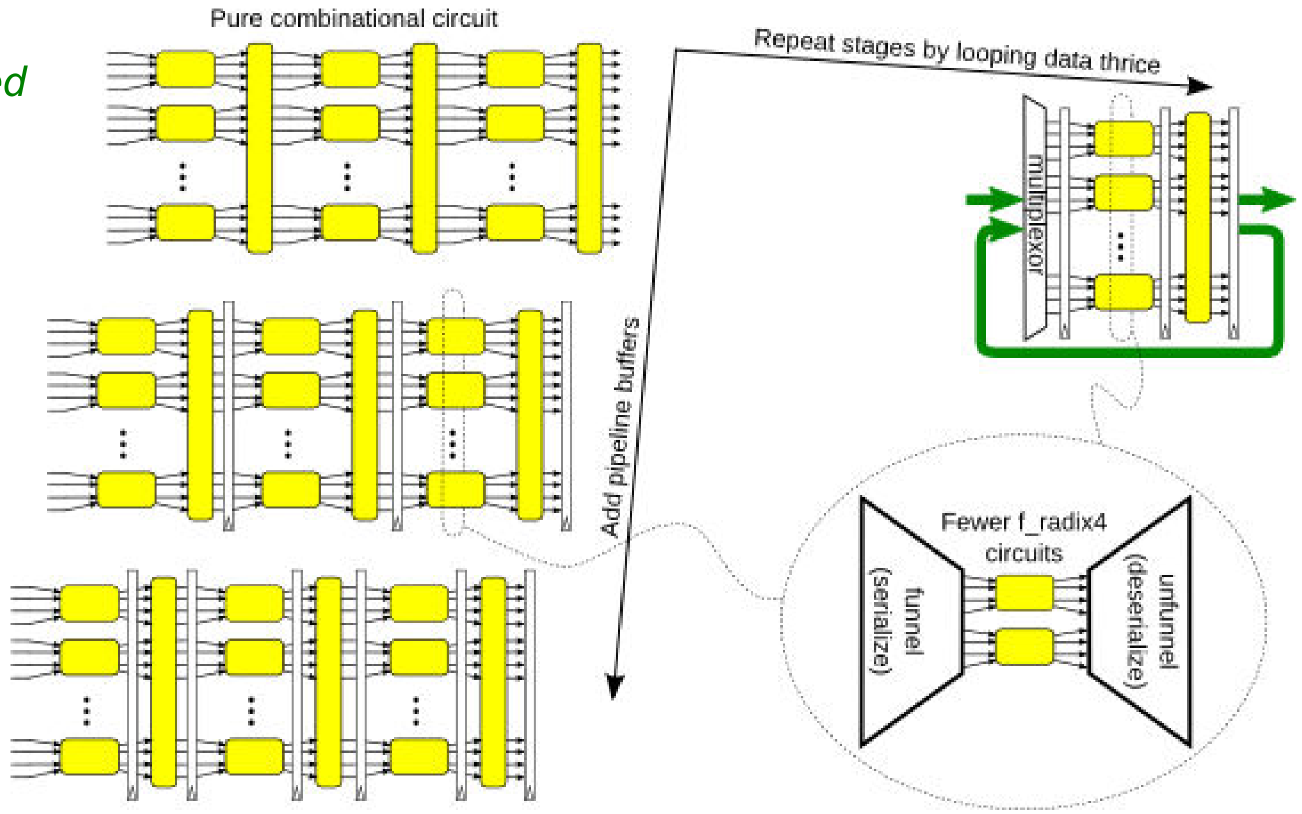
```
module [Module] mkIFFT (Server#(IFFTData, IFFTData));
   UInt #(2) jmax = 2;

   let s <- mkPipe_to_Server
              (  param_linear_not_looped
               ? mkLinearPipe_S (3, mkStage_S)
               : mkForLoop (jmax, mkStage_D, mkFn_to_Pipe (id)));
   return s;
endmodule
```

Results

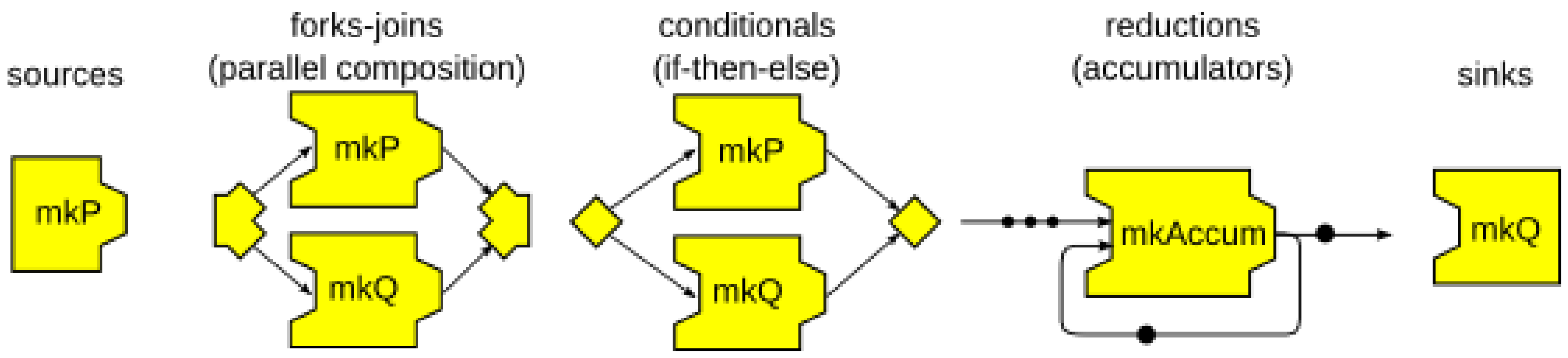
100 lines of BSV source code based on 4 parameters,
express all 24 architectures in the figure,
with a 10x variation in area/power

(which architecture is “best” depends on target
requirements, e.g., server vs. mobile)

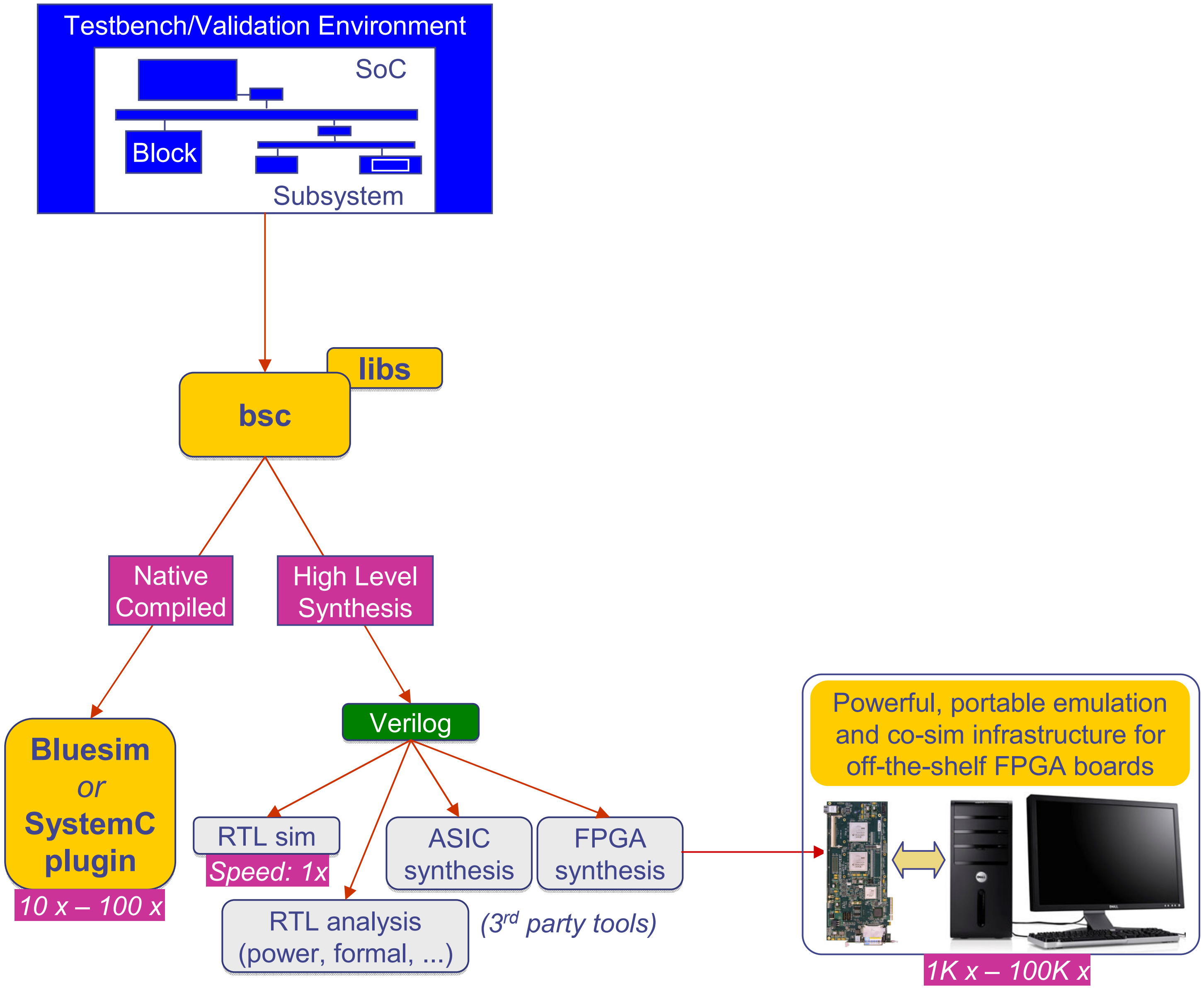
- *fully pipelined, flow-controlled*
- *all control logic*
  correct by construction



Other PAClib functions

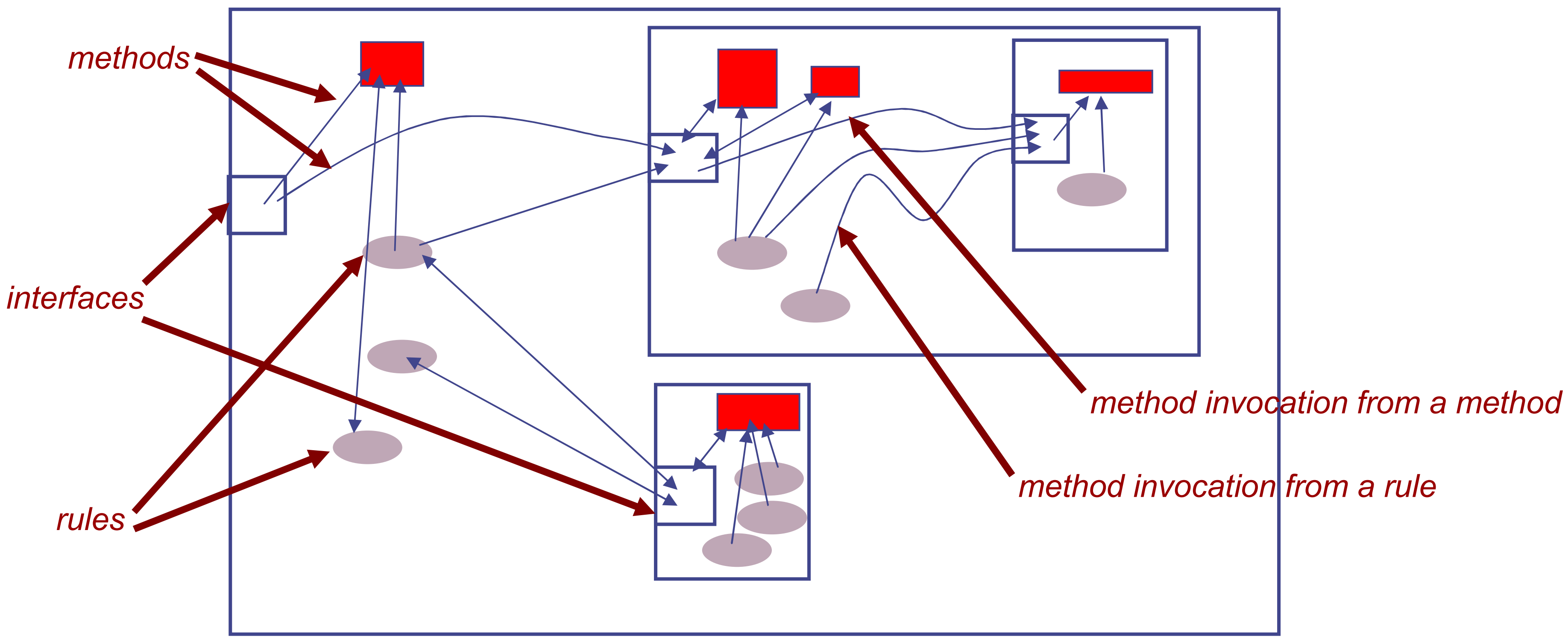


Synthesis/compilation and execution


Testbench/Validation Environment
SoC
Block
Subsystem
libs
bsc
Native Compiled
High Level Synthesis
Bluesim *or* SystemC plugin
10 x – 100 x
Verilog
RTL sim
Speed: 1x
ASIC synthesis
FPGA synthesis
RTL analysis (power, formal, ...)
(3rd party tools)
Powerful, portable emulation and co-sim infrastructure for off-the-shelf FPGA boards
1K x – 100K x

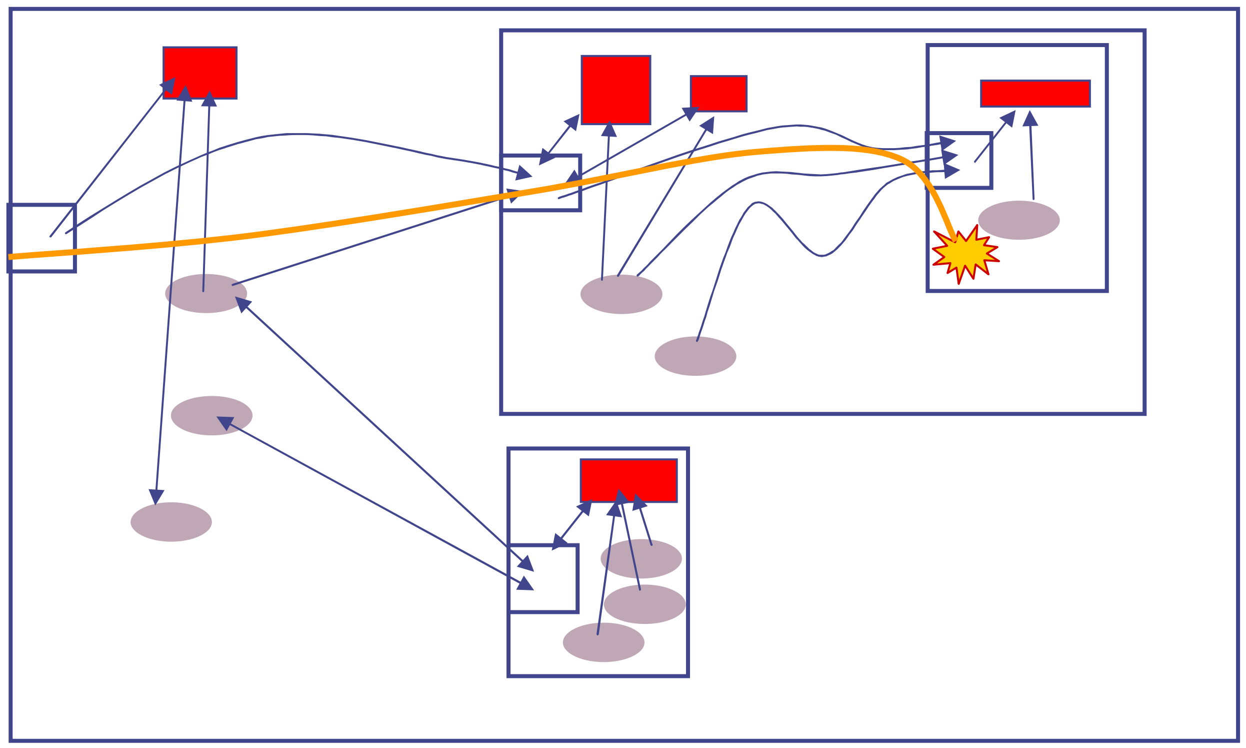
- PAClib: using HOFs for architectural flexibility
- HW Probes: using monads

BSV module hierarchy


methods
interfaces
rules
method invocation from a method
method invocation from a rule

Rules are <u>atomic</u> w.r.t. each other, even though they may reach across many modules.

Observing signals deep in the hierarchy



To observe a signal deep in the hierarchy, a set of wires must be brought out to the top.

In most HW design languages (e.g., Verilog, VHDL), this means adding this to the interface of all surrounding modules. Very messy and tedious.

BSV module structure

A module is actually a monad; this is executed during static elaboration of the program

```
module mkFoo #(... parameters ...)  (... interface ...);

   // ---- instantiation of sub-modules
   InterfaceType1  ifc1 <- mkBaz1 (... parameters ...);
   InterfaceType2  ifc2 <- mkBaz2 (... ifc1 ... parameters ...);
   ...



   // ---- RULES
   rule rl_A ( ... condition invoking methods in interfaces ... );
      ... rule body invoking methods in interfaces ...
   endrule

   ...

   // ---- INTERFACE DEFS
   method Bool isEmpty (... args ...)
   endmethod
   ...
endmodule
```

Collect sub-modules and rules

BSV module structure

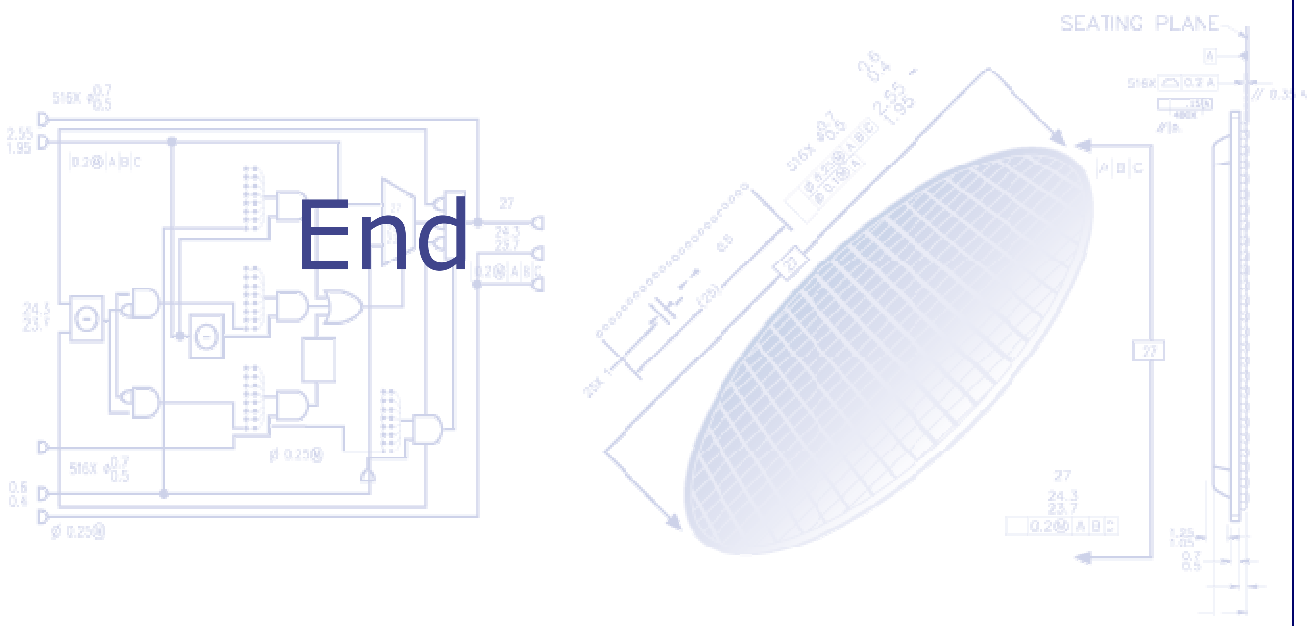
Use the monad to collect additional information

```
module mkFoo #(... parameters ...)  (... interface ...);

   // ---- instantiation of sub-modules
   InterfaceType1  ifc1 <- mkBaz1 (... parameters ...);
   InterfaceType2  ifc2 <- mkBaz2 (... ifc1 ... parameters ...);
   ...

   Probe #(type) probe <- mkProbe (... expression ...);

   // ---- RULES
   rule rl_A ( ... condition invoking methods in interfaces ... );
      ... rule body invoking methods in interfaces ...
   endrule

   ...

   // ---- INTERFACE DEFS
   method Bool isEmpty (... args ...)
   endmethod
   ...
endmodule
```

Collect sub-modules and rules and probe wires

All the probe “plumbing” is hidden

- PAClib: using HOFs for architectural flexibility
- HW Probes: using monads



End