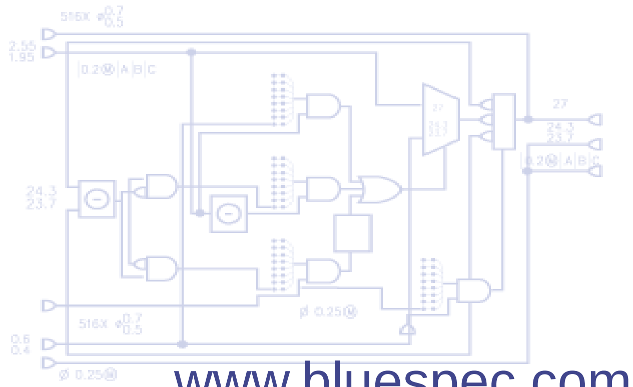
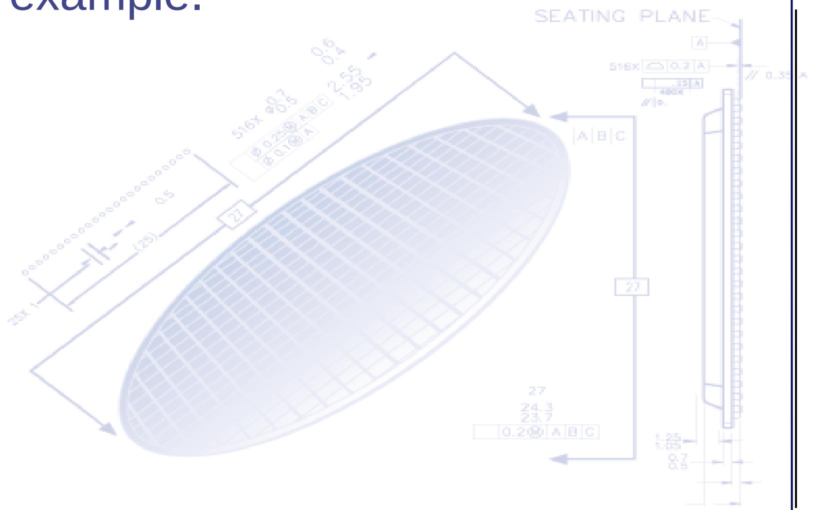
Bluespec Training

Eg03: Concurrent Bubblesort

A simple concurrent Bubblesort example.

```
import FIFO::*;

typedef Bit#(32) DataT;

module ex_in1_out2_bs(Empty);

  Integer fifo_depth = 16;

  function Bit#(1)   determine_queue(DataT val );
   return (val[0]);
  endfunction

  FIFO#(DataT) inbound1();
  mkSizedFIFO#(fifo_depth) the_inbound1(inbound1);
  FIFO#(DataT) outbound1();
  mkSizedFIFO#(fifo_depth) the_outbound1(outbound1);
  FIFO#(DataT) outbound2();
  mkSizedFIFO#(fifo_depth) the_outbound2(outbound2);

  rule enq1 (True);
     DataT in_data  = inbound1.first;
     FIFO#(DataT) out_queue  =
         determine_queue(in_data   ) == 0 ? outbound1 : outbound2;
     out_queue.enq(in_data    );
    inbound1.deq;
  endrule : enq1

endmodule  : ex_in1_out2_bs
```





www.bluespec.com

Generalization via 5 versions

The accompanying code demonstrates several versions:

Eg03a_Bubblesort/	Sorts 5 items, each of type `Int 32'. Completely sequential, to show correspondence with conventional software implementation.
Eg03b_Bubblesort/	Parallel version. Uses ‘maxBound’ (largest Int 32) as a “special” value different from any of the sorted values (like “+infinity”).
Eg03c_Bubblesort/	Generalizes ‘5’ items to ‘n’ items.
Eg03d_Bubblesort/	Generalizes items of type ‘Int 32’ to items of arbitrary type ‘t’ (i.e., makes the program polymorphic).
Eg03e_Bubblesort/	Uses the ‘Maybe t’ type to eliminate the need for a ‘maxBound’ special value.

Note: this is a pedagogical introductory example focusing on concurrency and modularity, and is not intended as an example of efficient sorting!

(See ‘Eg06_Mergesort’ for more efficient and scalable sorting.)

1st version: directory Eg03a_Bubblesort/

Examine the two source files: src/Top.bs src/Bubblesort.bs

We suggest that you take multiple passes through the files, incrementally increasing your understanding. Initially, just try to understand the syntactic structure, making frequent reference to the lecture slides in "Lec_Basic_Syntax" in the "Reference" directory.

The following excerpts show an outline of the syntactic structure of the two source files.

```
package Top where
…
import …
import Bubblesort
…
n :: Int 32 = 5
…
mkTop :: Module Empty
mkTop =
  module
    rg_j1 :: Reg (Int 32) <- mkReg 0
    …
    sorter :: Sort_IFC <- mkBubblesort
    …
    rules
```

```
package Bubblesort where
…
import …
…
interface Sort_IFC =
   …
mkBubblesort :: Module  Sort_IFC
mkBubblesort =
  module
    rg_j :: Reg (UInt 3) <- mkReg 0
    …
    rules
    …
    method definitions
```

1st version: directory Eg03a_Bubblesort/ (contd.)

Examine the two source files: src/Top.bs src/Bubblesort.bs

```
package Top where
…
import …
import Bubblesort
…
n :: Int 32 = 5
…
mkTop :: Module Empty
mkTop =
  module
    rg_j1 :: Reg (Int 32) <- mkReg 0
    …
    sorter :: Sort_IFC <- mkBubblesort
    …
    rules
```

```
package Bubblesort where
…
import …
…
interface Sort_IFC =
   …
mkBubblesort :: Module  Sort_IFC
mkBubblesort =
  module
    rg_j :: Reg (UInt 3) <- mkReg 0
    …
    rules
    …
    method definitions
```

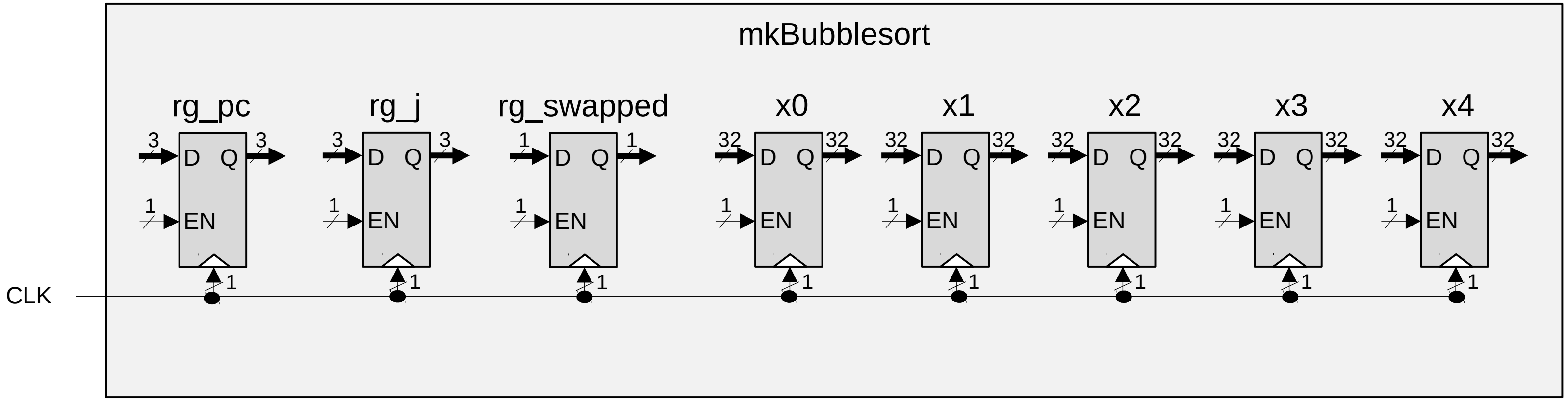
Notes on the syntactic structure:

- The top-level module is mkTop; it has an Empty interface (an interface with no methods).
- It instantiates several sub-modules: a few registers, a pseudo-random number generator (LFSR, for Linear Feedback Shift Register), and the mkBubblesort module.
- The mkBubblesort module, in turn, instantiates some registers as its sub-modules.
- The rules in the Top invoke the put and get methods in the bubblesort module's interface.
- All the rules and methods have Boolean conditions indicating when they are "ready".

1st version: directory Eg03a_Bubblesort/ (contd.)

In module mkBubblesort:

- The register rg_pc is used to sequence the sorting actions. The name is suggestive of “Program Counter”. 3 bits are enough to distinguish all the states.
- The register rg_j is used to count 5 incoming values. Thus, it is typed UInt 3, i.e., an unsigned integer of 3 bits, capable of holding values 0..7. It’s reset value (initial value) is 0.
- The register rg_swapped remembers whether any swap occurred in the current pass.
- The registers x0..x4 hold the 5 values to be sorted. Each value is of type Int 32, i.e., a signed integer of 32 bits. Each register’s reset value is maxBound, a symbolic name representing the largest Int 32, i.e., $+2^{31-1}$. We assume all input values to be sorted will be strictly < maxBound



The module mkBubblesort, showing the registers in isolation

1st version: directory Eg03a_Bubblesort/ (contd.)

The rules in module mkBubblesort essentially encode a sequential algorithm similar to that shown below in a pseudo-C notation:

```
while (True)
    swapped = False;
    for (pc = 1; pc < 5; pc++) {
        if (x[pc-1] > x[pc] {
            swap them;
            swapped = True;
        }
    if (! swapped) break;
}
```

Each rule has:

- A condition that says when it can fire, and
- A body that says what happens if it fires.

Building and running the codes

Each variation is built and run in the same way:

- In the Build/ directory you can use the ‘Makefile’ for building and running Bluesim or Verilog sim:

```
$ make  b_compile  b_link  b_sim          // for Bluesim
$ make  v_compile  v_link  v_sim          // for Verilog sim
```

Note: When building for Bluesim, compiler-temporary files are created in the build_bsim/ directory.
When building for Verilog, temporaries are in build_v/ and Verilog files are in verilog_dir/.
‘make clean’ and ‘make full_clean’ will clean up these directories.

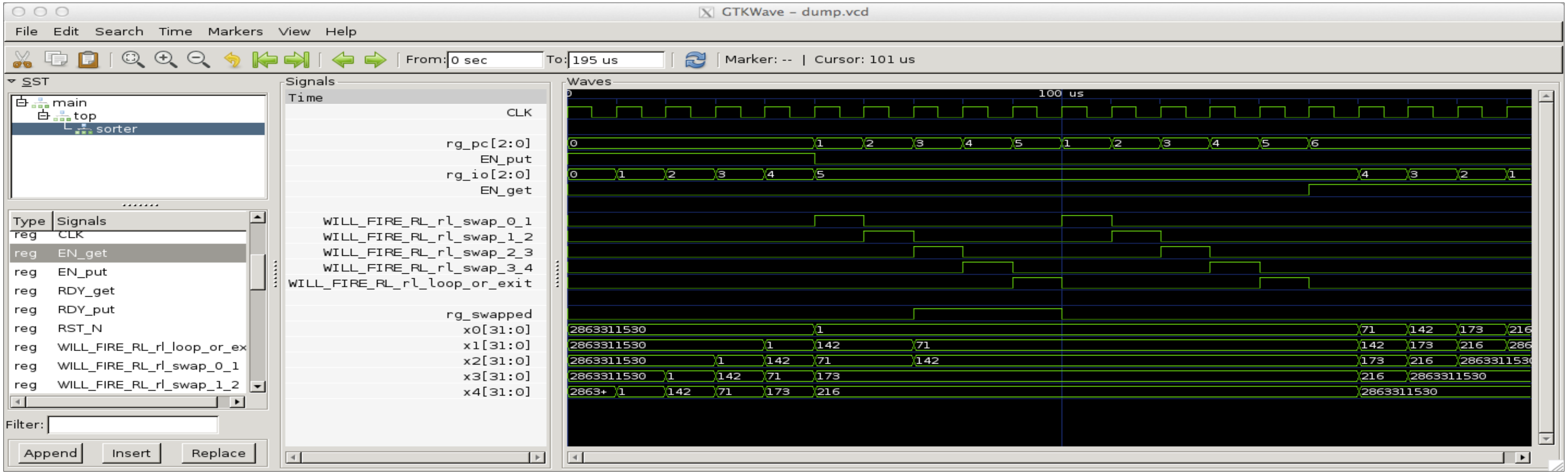
Build and run the 1st version

- In the Build/ directory, practice building and running the program in both ways:
  - Using ‘make’ commands, for Bluesim.
  - Using ‘make’ commands, for Verilog sim.

- Observe the inputs and outputs and verify that they are reasonable (there are 5 inputs and outputs, and the outputs are a sorted version of the inputs).

- When you run your simulation, a file “dump.vcd” is created, containing waveforms, which you can view in your favorite waveform viewer.
  - (It is created because of the –V flag for Bluesim or the +bscvcd flag for Verilog sim.)
  - The picture “Waves_screenshot.tiff” is a screenshot of such a view.
  - Study the waveform and make sure you understand how it reflects the behavior of the Bluespec code.

Waveforms from the 1st version

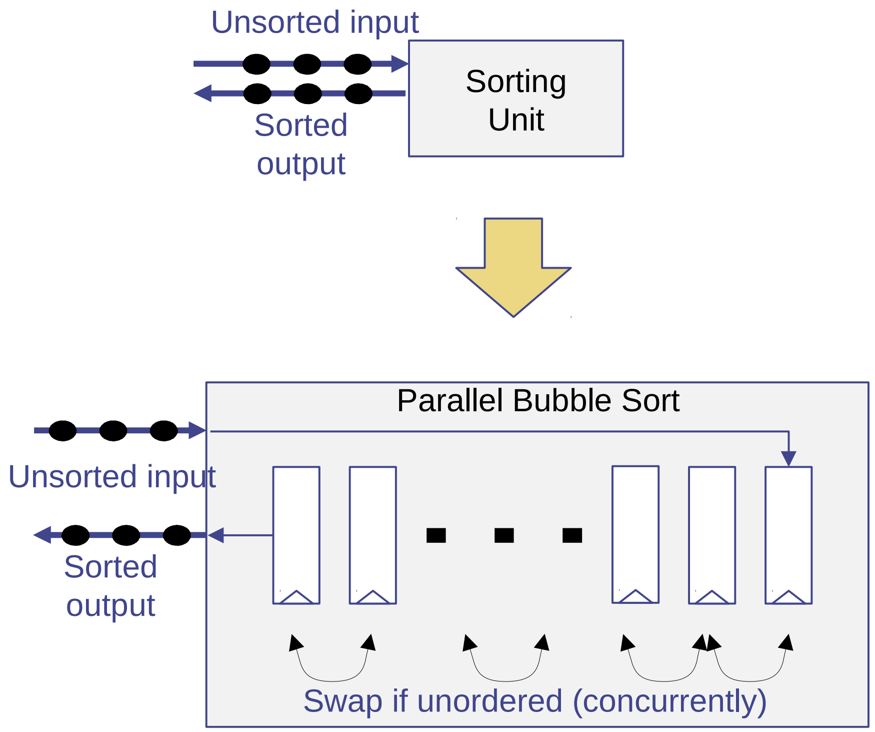


```
main
  top
    sorter

Type  Signals
reg   CLK
reg   EN_get
reg   EN_put
reg   RDY_get
reg   RDY_put
reg   RST_N
reg   WILL_FIRE_RL_rl_loop_or_ex
reg   WILL_FIRE_RL_rl_swap_0_1
reg   WILL_FIRE_RL_rl_swap_1_2
```

```
Time                        0                                   100 us
CLK
rg_pc[2:0]                  0  1 2 3 4 5 1 2 3 4 5 6
EN_put
rg_io[2:0]                  0 1 2 3 4 5  4 3 2 1
EN_get
WILL_FIRE_RL_rl_swap_0_1
WILL_FIRE_RL_rl_swap_1_2
WILL_FIRE_RL_rl_swap_2_3
WILL_FIRE_RL_rl_swap_3_4
WILL_FIRE_RL_rl_loop_or_exit
rg_swapped
x0[31:0]                    2863311530  1  71 142 173 216
x1[31:0]                    2863311530  1 142 71  142 173 216 286
x2[31:0]                    2863311530  1 142 71 142  173 216 286331153
x3[31:0]                    2863311530  1 142 71 173  216 2863311530
x4[31:0]                    2863+ 1 142 71 173 216  2863311530
```

bluespec

Eg03: Basic concurrency and modularity



Algorithm/Architecture for remaining versions:

- The module accepts a stream of ‘n’ input items (unsorted).
- These are shifted in to ‘n’ registers.
- Concurrently (and even while inputs are arriving), whenever two adjacent registers contain values in the wrong order, we swap their contents.
- When ‘n’ inputs have been received, and all of them are sorted, the module yields a stream of ‘n’ sorted outputs by shifting them out.
- When ‘n’ outputs have been streamed out, the module is ready for its next set of ‘n’ inputs.

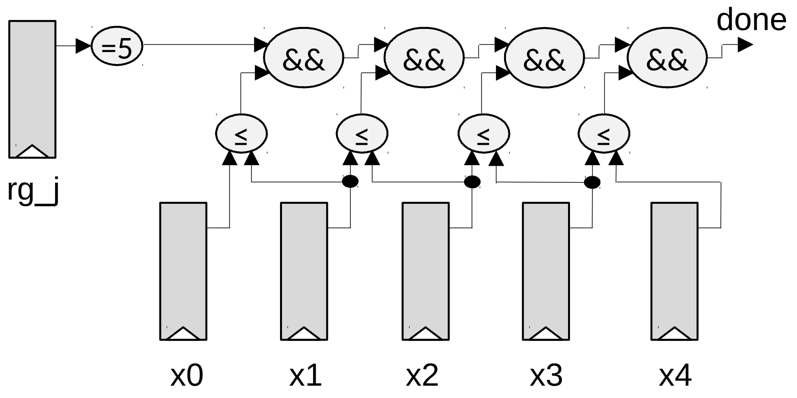
2nd version: directory Eg03b_Bubblesort/

In module mkBubblesort:

- The Bool variable “done” defines the condition that all values have been received and are sorted:

```
    -- Test if array is sorted
    let done :: Bool
        done = ((rg_inj == 5) && (x0 <= x1) && (x1 <= x2) && (x2 <= x3) && (x3 <= x4))
```

- In Bluespec, value-expressions like this just represent combinational circuits:



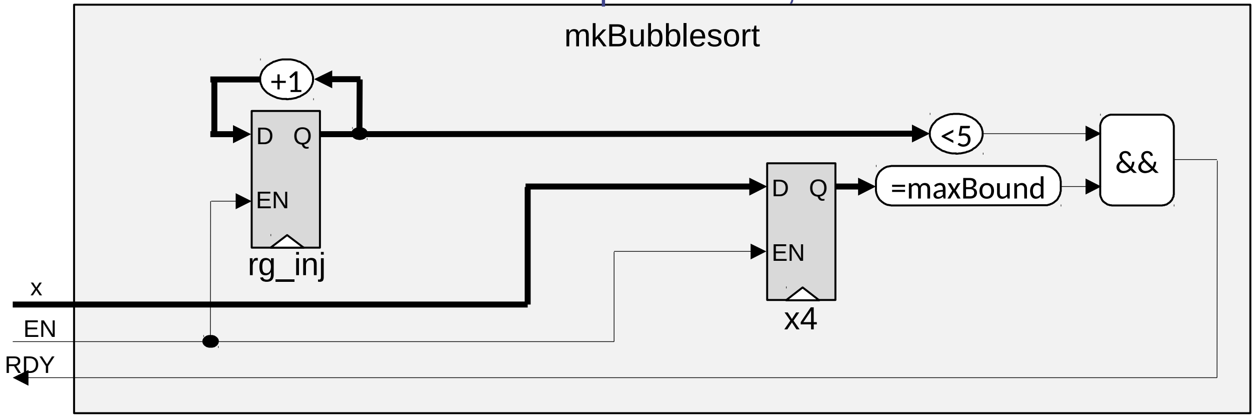
2nd version: directory Eg03b_Bubblesort/ (contd.)

```
-- Inputs: feed input values into x4
put :: Int  32 -> Action
put x = do
            x4 := x
            rg_inj := rg_inj + 1
        when ((rg_inj < 5) && (x4 == maxBound))
```

In module mkBubblesort:

- The "put" method is of type Action, i.e., it's body is an expression of type Action.
  - The body has two sub-Actions: one places a new input value x into register x4, and the other increments the input count rg_j.
  - The method condition ensures two things:
    - "(rg_j < 5)" ensures that we stop after receiving 5 inputs
    - "(x4==maxBound)" ensures that we don't over-write a meaningful value already in x4, i.e., it can place a new value in x4 only after the swap rules have moved the previous value out and replaced it with maxBound.

Hardware for put method, in isolation



2nd version: directory Eg03b_Bubblesort/ (contd.)

```
-- Outputs: drain by shifting them out of x0
get :: ActionValue (Int  32)
get = do
        x0 := x1
        x1 := x2
        x2 := x3
        x3 := x4
        x4 := maxBound
        if1 (x1 == maxBound) (rg_inj := 0)
        return x0
      when  done
```

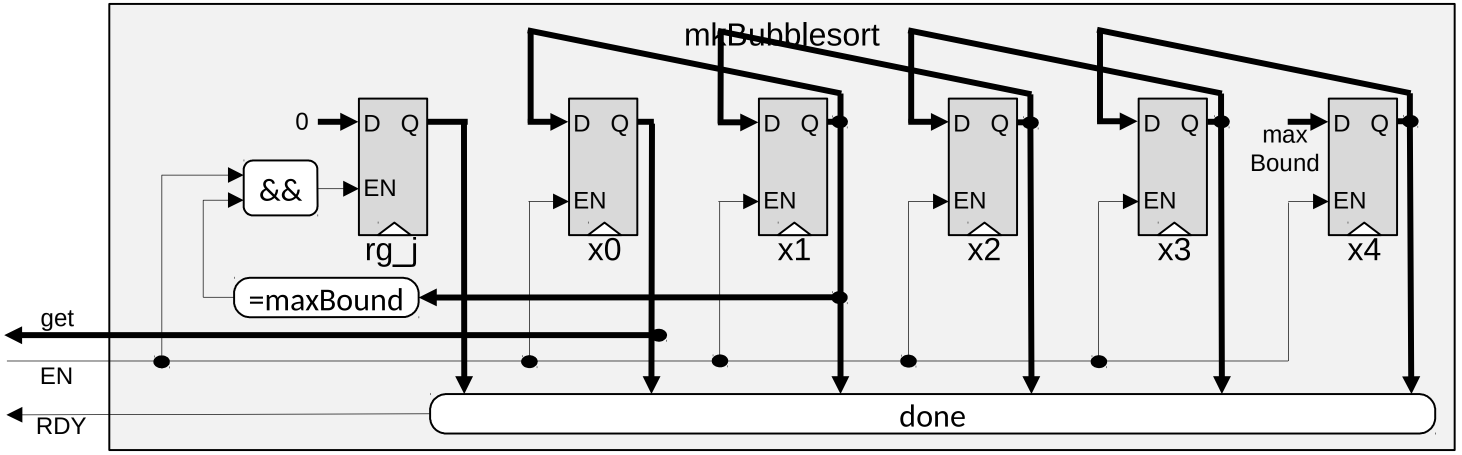
In module mkBubblesort:

- The "get" method is of type ActionValue (Int 32), i.e., it's body is an Action, and it also returns a value of type Int 32.
  - The body returns x0, shifts all remaining values in x1..x4 down into x0..x3, respectively, and shifts the value maxBound into x4.
  - The method condition of the "get" method:
    - Is only enabled when "done" is true, i.e., all inputs have been received and all values are sorted
    - Is only enabled until we've returned the 5th value, after which x0 contains maxBound
  - When we return return the 5th value (from x0), x1 will be maxBound; at this time we can reset rg_j to 0
  - Note that after returning 5 values, the module is again in it's original state—rg_j contains 0, and x0..x4 contain maxBound—and so it is ready to receive the next 5 values to be sorted

2nd version: directory Eg03b_Bubblesort/ (contd.)

```
      -- Outputs: drain by shifting them out of x0
      get :: ActionValue (Int  32)
      get = do
              x0 := x1
              x1 := x2
              x2 := x3
              x3 := x4
              x4 := maxBound
              if1 (x1 == maxBound) (rg_inj := 0)
              return x0
            when  done
```

Hardware for "get" method, in isolation



bluespec

2nd version: directory Eg03b_Bubblesort/ (contd.)

```
"rl_swap_0_1": when  (x0 > x1) ==> do
    x0 := x1
    x1 := x0

"rl_swap_1_2": when  (x1 > x2) ==> do
    x1 := x2
    x2 := x1
```

```
"rl_swap_2_3": when  (x2 > x3) ==> do
    x2 := x3
    x3 := x2

"rl_swap_3_4": when  (x3 > x4) ==> do
    x3 := x4
    x4 := x3
```

In module mkBubblesort:

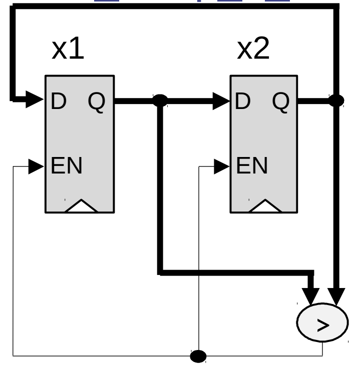
- In each of the four rules:
  - The rule condition (an expression of type Bool), e.g., “(x0>x1)” tests if two adjacent values are in the wrong order. This is a prerequisite for the rule to “fire”, i.e., to execute its body.
  - The rule body (an expression of type Action) is composed of two sub-Actions whose effect is to swap the contents of the corresponding two registers. Note:
    - Expressions/methods/functions of type “Action” or “ActionValue t” (potentially) change the state of the system (e.g., write to a register, write to memory, enqueue onto a FIFO, etc.). We also say that such expressions have “side-effects”.
    - Expressions that are not of type Action or ActionValue (e.g., Bool) *never* change the state of the system (this is guaranteed by Bluespec’s type-checking rules). We also say that these are “pure” or “value” expressions.
    - All Actions in a rule are considered instantaneous and simultaneous. The two assignments are not read sequentially (as is typical in software programs), and this explains why there is no need for a “temporary” intermediate variable (as is typical in software programs)

2nd version: directory Eg03b_Bubblesort/ (contd.)

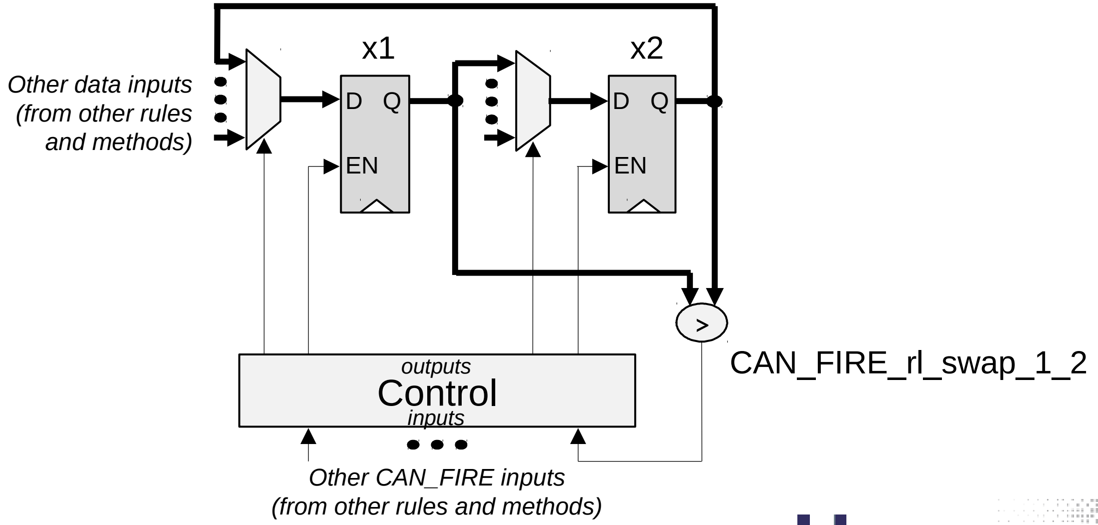
```
"rl_swap_0_1": when  (x0 > x1) ==> do
    x0 := x1
    x1 := x0

"rl_swap_1_2": when  (x1 > x2) ==> do
    x1 := x2
    x2 := x1
```

```
"rl_swap_2_3": when  (x2 > x3) ==> do
    x2 := x3
    x3 := x2

"rl_swap_3_4": when  (x3 > x4) ==> do
    x3 := x4
    x4 := x3
```

Hardware for rl_swap_1_2, in isolation



… and in more detail



2nd version: directory Eg03b_Bubblesort/ (contd.)df

In module mkTop:

- rg_j1 and rg_j2 are used to count 5 inputs and 5 outputs respectively. They are somewhat arbitrarily typed at Int 32 (since they count 0..4, even Int 3 or UInt 3 would have been ok, but we usually don't worry so much about minimizing state bits in a testbench).
- mkLFSR_8 instantiates, from the Bluespec library, a pseudo-random number generator of 8-bit values (see Sec.C.8.1 in the Reference Guide).

- In rule "rl_feed_inputs":
  - The rule condition ensures that is only enabled to generate 5 inputs
  - lfsr.value() is a "value method" that returns a Bit 8; zeroExtend() extends this to Bit 32; unpack() converts this to an Int 32. lfsr.next() is an Action method that tells the LFSR to advance to its next random number
  - "sorter.put(x)" sends this new input into the bubblesort module
    - Recall that the "put" method in mkBubblesort has a method condition that prevents overwriting a previous value. Thus, rl_feed_inputs can only fire when that condition is true.

- In rule "rl_drain_outputs"
  - The rule condition ensures that it only retrieves 5 outputs
  - "sorter.get()" yields the next output, whenever the method condition on the "get" method allows, and this value is displayed.

2nd version: directory Eg03b_Bubblesort/ (contd.)

Summary of behavior:

- Rule “rl_feed_inputs” in the testbench module feeds 5 random values into the sorter module using the “put” method.

- The sorter module receives the 5 inputs via the “put” method. Sorting begins as soon as the first input arrives (the first step sorts the first input against its +infinity neighbor), and continues as more inputs arrive. Sorting may continue for some time after all inputs arrive, until all 5 registers are sorted.

- When all 5 registers are sorted, the “get” method is enabled, and the testbench drains out the 5 values in order. The sorter module yields these 5 values in order by shifting them through the registers.

- After yielding 5 outputs, the sorter module is back in its initial state, and ready for another 5 inputs (although our testbench quits after the first set).

This behavior can seen in the waveforms for this design created during simulation. We will shortly describe simulation and waveform generation but, for your convenience, the directory includes a screen shot (“waves_screenshot.tiff”) which you can view directly.

Build and run the 2nd version

- In the Build/ directory, practice building and running the program in both ways:
  - Using ‘make’ commands, for Bluesim.
  - Using ‘make’ commands, for Verilog sim.

- Observe the inputs and outputs and verify that they are reasonable (there are 5 inputs and outputs, and the outputs are a sorted version of the inputs).

- Note that an input is supplied only on *every other* clock. Why?
  - (Hint: see previous remarks on method condition of the “put” method.)

- When you run your simulation, a file “dump.vcd” is created, containing waveforms, which you can view in your favorite waveform viewer.
  - (It is created because of the –V flag for Bluesim or the +bscvcd flag for Verilog sim.)

  - The picture “waves_screenshot.tiff” is a screenshot of such a view. Note that it shows Bluespec source code types Int 32, Bool, …, and not raw Verilog bits.

  - Study the waveform and make sure you understand how it reflects the behavior of the Bluespec code.

Suggested exercises

In this and future examples, we suggest extra exercises to deepen your understanding of Bluespec, which you can pursue on your own (may not be covered during classroom training)

- Examine the generated Verilog file mkBubblesort.v (in the “verilog_RTL/” directory).
    - Look at the input and output ports, and understand how they correspond to the Bluespec interface Sort_IFC and its methods.
    - Skim the interior of the Verilog module, and notice correspondences with the Bluespec source module (registers, rules, rule and method conditions, …).
- Analyze and explain in detail what would happen if the testbench supplied ‘maxBound’ as one (or more) of the input values. If you wish, modify the testbench to test this.
- Modify the testbench to sort two (or three, or more) rounds of 5-input sequences, instead of quitting after the first round.
- Modify the testbench and the bubblesort modules to sort sequences of some other length instead of 5 (say, 10).
- When you compile, you'll see messages that a rule was treated as “more urgent” than another.
    - Look at the Makefile, and you'll see that by default it's compiled with OPTION1
    - Try it with OPTION2, OPTION3 and OPTION4, and understand the code for them

Time-out to reinforce some concepts

Before moving on with the examples, please study the lecture: Lec_Rule_Semantics
to understand the concepts behind Rules:

- Semantics of individual rules (and the methods they call)
  - “simultaneous/parallel actions”, “instantaneous”

- Semantics of concurrency of a set of rules in each clock
  - “concurrency”
  - “ordering constraints” between methods in different rules
  - “rule schedules”

3rd version: directory   Eg03c_Bubblesort/

The 3rd version generalizes the 2nd version from sorting 5 numbers to sorting *n* numbers

Examine the two source files: src/Top.bs   src/Bubblesort.bs
It is useful to compare these side-by-side with the corresponding two files in Eg03b_Bubblesort/src/, to see how they have changed.

Notice that the interface type is now parameterized by *n*:

```
interface (Sort_IFC :: # -> *)  n_t =
```

In Bluespec, certain types and type parameters can be *numeric types (of "kind" #).*

Note that *numeric types* (which are only meaningful at compile time) are distinct from *numeric values* (which can of course occur at run time).
  Bluespec is very strict in type-checking—there are no automatic conversions
  The code uses these explicit conversions:

$$\begin{tikzcd}\text{numeric type} \arrow[r, "\text{valueOf()}"] & \text{Integer} \arrow[r, "\text{fromInteger()}"] & \text{UInt 16}\end{tikzcd}$$

3rd version: directory Eg03c_Bubblesort/ (contd.)

The 2nd version's explicit registers x0..x4 have here been replaced by a vector of registers:

```
    -- A vector of registers to hold the values being sorted
    -- Note: 'maxBound' is largest 'Int 32'; we assume none of the
    -- actual values to be sorted have this value.
    xs :: Vector  n_t  (Reg  (Int  32)) <- replicateM  (mkReg  maxBound)
```

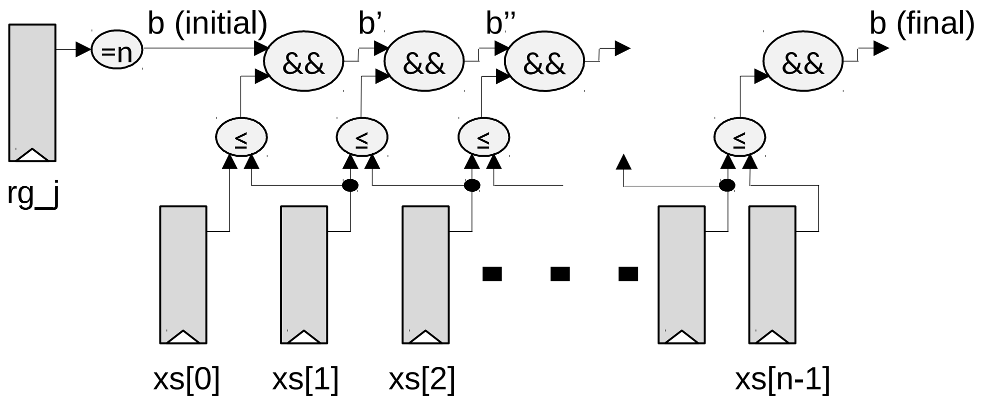
"replicateM" repeatedly applies "mkReg(maxBound)" to create them.

3rd version: directory Eg03c_Bubblesort/ (contd.)

The 2nd version's explicit "done" test is here computed using List.all and a lambda-expression predicate:

```
-- Test if array is sorted
let done :: Bool
    done = (   (rg_inj == fromInteger  n)
            && (List.all (\i -> ((xs !! i)._read <= (xs !! (i+1))._read))
                         (List.upto  0  (n - 2))))
```

Note: this List.all "loop" is also "statically elaborated" (i.e., unfolded by the compiler) into a circuit of combinational logic



Observe that, unlike traditional languages, there is no storage location corresponding to loop variables like i and the accumulated value (call it "b"). The former has disappeared completely, and the latter exists in multiple versions (b, b', b'', …) each of which just names a wire.

3rd version: directory Eg03c_Bubblesort/ (contd.)

The 2nd version's separately-written rules are here replaced by a List.map loop generating them:

```
        -- Function to generate rule to swap xs[i] and xs[i+1] if unordered
        gen_swap_rule :: Integer -> Rules
        gen_swap_rule  i = let
                               xs_i        = xs !! i
                               xs_i_plus_1 = xs !! (i+1)
                           in
                               rules
                                 when (xs_i > xs_i_plus_1) ==> do
                                     xs_i        := xs_i_plus_1
                                     xs_i_plus_1 := xs_i

    -- Add the rules to the module in descending urgency order
    addRules_descending_urgency  (List.map  gen_swap_rule  (List.upto  0  (n - 2)))
```

Note: this for-loop is "statically elaborated" (i.e., unfolded by the compiler), and is equivalent to writing out n-1 rules explicitly.

3rd version: directory Eg03c_Bubblesort/ (contd.)

In the “get” method, the following line:

```
writeVReg  xs  (shiftInAtN  (readVReg  xs)  maxBound)
```

uses a number of Bluespec library functions on vectors:

- readVReg() returns a vector of values corresponding to the contents of a vector of registers
- shiftInAtN() takes a vector of values x1,..,xN and a new value y, and returns a vector of values x2,..,xN,y (discards x1)
- writeVReg() writes a vector of values into a vector of registers

Note: this “loop” is also “statically elaborated” (i.e., unfolded by the compiler)

Time-out to reinforce some concepts

Before moving on with the examples, please study the lecture: Lec_Types
to understand the concepts behind types, polymorphism, and numeric types.

Please also *skim* through Section C.3 (“Vectors”) in the Reference Guide.

Synthesis hierarchy

- In the 1st and 2nd versions, in front of 'module mkBubblesort' line, there is a {-# verilog...#-} attribute:

```
{-# verilog mkBubblesort #-}

mkBubblesort :: Module  Sort_IFC
mkBubblesort =
  module
```

- When generating Verilog, this creates a 'mkBubblesort.v' file with a 'mkBubblesort' Verilog module

- In the 3rd version, this {-# verilog...#-} attribute is removed.
- This is because Bluespec cannot separately synthesize *polymorphic* modules; they can only be inlined into a parent module
- Instead, in Top.bs, we have created a specific instance of the module (for *n*=20); since this is no longer polymorphic, it can be separately synthesized. This is the module actually instantiated in the module mkTop:

```
-- Instantiate and separately synthesize a Bubblesort module for size 'N_t'

{-# verilog mkBubblesort_nt #-}

mkBubblesort_nt :: Module  (Sort_IFC  N_t)
mkBubblesort_nt =
  module
    m :: Sort_IFC  N_t <- mkBubblesort
    return m
```

- This is a common idiom in Bluespec, for creating a separately synthesized instance of a polymorphic module

Build and run the 3rd version

- In the Build directory, build and run using the ‘make’ commands, with Bluesim and/or with Verilog sim, as described earlier
- Observe the inputs and outputs and verify that they are reasonable (that there are 20 inputs and outputs, and the outputs are a sorted version of the inputs)
- In Top.bs, change the 1 line that defines the size of the problem:

```
type  N_t = 20
```

Rebuild and re-run to test it.

4th version: directory Eg03d_Bubblesort/

The 4th version generalizes the 3rd version from sorting values of type Int 32 to sorting values of any type “t”

Examine the two source files: src/Top.bs src/Bubblesort.bs

It is useful to compare these side-by-side with the corresponding two files in the previous version, to see how they have changed.
Notice that the interface type is now further parameterized by *t*:

```
interface (Bubblesort_IFC :: # -> * -> *)  n_t  t =
```

The mkBubblesort type declaration is now extended with *type contexts* (we also call them *provisos*):

```
mkBubblesort :: (Bits  t  wt,                -- ensures 't' has a hardware bit representation
                 Ord  t,                     -- ensures 't' has the '<=' comparison operator
                 Eq  t,                      -- ensures 't' has the '==' comparison operator
                 Bounded  t)                 -- ensures 't' has a 'maxBound' element
                 =>
                 Module (Bubblesort_IFC  n_t  t)
```

These are assertions, respectively, that:

- values of type t have a bit representation (with bit-width wt), since we need to store them in registers
- values of type t can be compared with the <= operator (ordering)
- values of type t can be compared with the == operator (equality)
- there exists a ‘maxBound’ value of type t

Without these assertions, the compiler would emit a type-checking error, since all these properties are used inside the module.

Time-out to reinforce some concepts

Please study the lecture: Lec_Typeclasses

to understand the concepts behind Bits, Ord, Eq, Bounded.

Build and run the 4th version

- In the Build directory, build and run using the ‘make’ commands, with Bluesim and/or with Verilog sim, as described earlier
- Observe the inputs and outputs and verify that they are reasonable (that there are 20 inputs and outputs, and the outputs are a sorted version of the inputs)
- In Top.bs, change the two lines that define the size of the problem and the type of values being sorted:

```
type N_t = 20
type MyT = UInt  24
```

Rebuild and re-run to test it.

5th version: directory Eg03e_Bubblesort/

The 5th version eliminates the dependency on the existence of a separate ‘maxBound’ value that is separate from legitimate input values.

Before proceeding, please review the lecture: Lec_Types
and, specifically, the section on “Maybe” types.

Examine the two source files: src/Top.bs src/Bubblesort.bs

It is useful to compare these side-by-side with the corresponding two files in the previous version, to see how they have changed.

Notice that the “Bounded t” proviso has been removed from module mkBubblesort.

The vector of registers now contain values of type “Maybe t” instead of “t”:

```
    -- A vector of registers to hold the values being sorted
    xs :: Vector  n_t  (Reg  (Maybe  t)) <- replicateM  (mkReg  Invalid)
```

These are initialized/reset to the value “Invalid”.
This plays the role previously played by “maxBound”.

In the “put” method, instead of inserting x into the register, we insert:

```
                xs !! jMax := Valid x
```

5th version: directory Eg03e_Bubblesort/ (contd.)

After the module mkBubblesort, there is some new code

```
instance (Ord  t) => Ord  (Maybe  t) where
   (<=) :: (Maybe  t) -> (Maybe  t) -> Bool
   mx1 <= mx2 = case (mx1, mx2) of
   …
```

A value of type Maybe t has two forms:

- Invalid
- Valid x

Conceptually, if values of type t are represented in n bits, then a value of type Maybe t is represented in n+1 bits. The extra bit is called a tag. When the tag is 0 (Invalid), the remaining n bits don't matter. When the tag is 1 (Valid), the remaining n bits represent a legitimate value of type t.

The "instance" declaration defines how to compare two values of type Maybe t:

- Invalid <= Invalid
- Valid x < Invalid for any x
- Invalid > Valid y for any y
- Valid x <= Valid y if (x <= y)

Please see the lecture: Lec_Typeclasses
for a more detailed explanation of these concepts.

Build and run the 5th version

- In the Build directory, build and run using the ‘make’ commands, with Bluesim and/or with Verilog sim, as described earlier

- Observe the inputs and outputs and verify that they are reasonable (that there are 20 inputs and outputs, and the outputs are a sorted version of the inputs)

- In Top.bs, change the two lines that define the size of the problem and the type of values being sorted:

```
type N_t = 20
type MyT = UInt  24
```

Rebuild and re-run to test it.

Time-out to reinforce some concepts

Please study the lecture: Lec_Types
to understand the concepts behind the Maybe type.

Suggested exercises

- Change the program to sort in *descending* order instead of ascending order.

- Instead of sorting scalar values, change Top to sort struct values:
  - Define a struct type with at least two fields.
  - Generate structs with random values for these fields.
  - Define the "<" operator on this struct type so as to compare only one field (you will have to declare an "Ord" instance for this struct type).
  - Test your program.

- Study Section C.3.10 ("Fold functions") in the Reference Guide. In the sorter module, redefine the "done" function to use foldl(), foldr() or fold() in place of the explicit for-loop. What is the circuit structure using the for-loop, foldl, foldr and fold? What is the advantage of the circuit structure using fold?

- The "done" function is a combinational circuit (using any of the above implementations). For large *n*, this can limit the clock speed at which the circuit can be synthesized. Split the circuit into multiple smaller stages, by adding registers at suitable points and introducing rules to propagate values.
  - This means that detecting "done" will be delayed by a few cycles. Does this matter?
  - How should you reset these registers during final output in the "get" method?

Summary

These examples have provided a basic familiarity with many of the concepts in the Bluespec language:

- File and package structure
- Module structure, module instantiation, interfaces and methods
- Rules and methods, and their semantics
- Types and typeclasses, numeric types
- Static elaboration
- Parameterization on sizes
- Parameterization on types (polymorphism)

It has also provided practice in using various Bluespec tools:

- Building and executing in Bluesim
- Generating Verilog, and executing in Verilog sim
- Using Makefiles
- Generating waveforms and viewing them

End

```
import FIFO::*;

typedef Bit#(32) DataT;

module ex_in1_out2_bs(Empty);

  Integer fifo_depth = 16;

  function Bit#(1)   determine_queue(DataT val );
   return (val[0]);
  endfunction

  FIFO#(DataT) inbound1();
  mkSizedFIFO#(fifo_depth) the_inbound1(inbound1);
  FIFO#(DataT) outbound1();
  mkSizedFIFO#(fifo_depth) the_outbound1(outbound1);
  FIFO#(DataT) outbound2();
  mkSizedFIFO#(fifo_depth) the_outbound2(outbound2);

  rule enq1 (True);
      DataT in_data  = inbound1.first;
      FIFO#(DataT) out_queue  =
          determine_queue(in_data    ) == 0 ? outbound1 : outbound2;
      out_queue.enq(in_data    );
     inbound1.deq;
   endrule : enq1

endmodule  : ex_in1_out2_bs
```