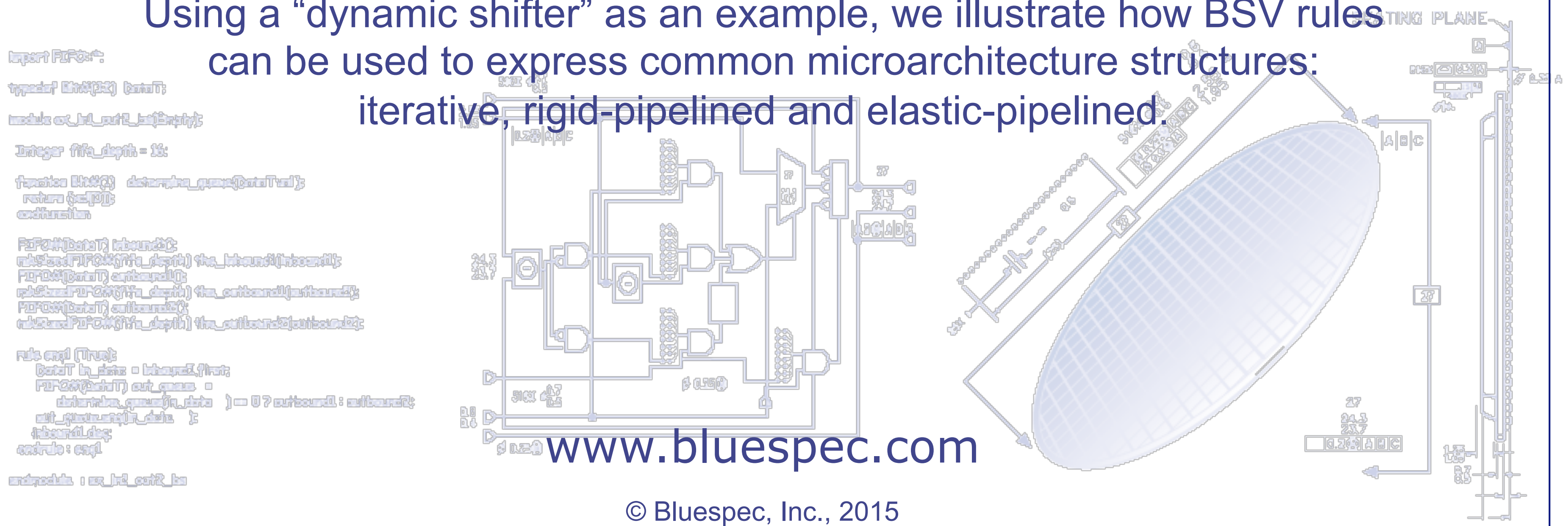
BSV Training

Eg04: Microarchitectures: FSMs and Pipelines

Using a “dynamic shifter” as an example, we illustrate how BSV rules can be used to express common microarchitecture structures: iterative, rigid-pipelined and elastic-pipelined.



www.bluespec.com

Dynamic shifts

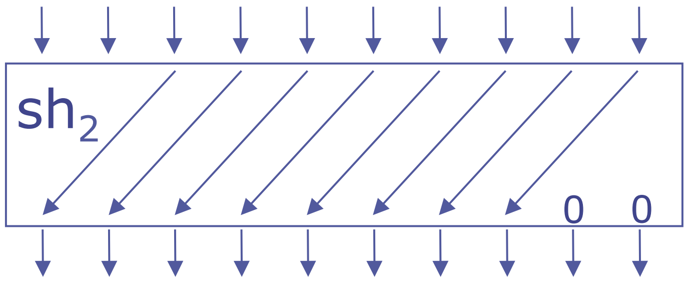
- Goal: circuit to implement a left-shift by a dynamic amount: z = shift (x, y)
  i.e., z = x left-shifted by y positions, where y is dynamic (run-time value)

- Algorithm: a dynamic shift can be achieved as a composition of static shifts corresponding to each bit of y.

- Example: Suppose y has type Bit #(3)
  - shift (x,y) =

shift x	by 1	$(= 2^0)$	if y[0] == 1
and	by 2	$(= 2^1)$	if y[1] == 1
and	by 4	$(= 2^2)$	if y[2] == 1

- Note: shifting by constant 2^J is trivial: just a "lane change" using only wires, no gates:



Example variations

The accompanying code demonstrates six variations:

- The following are in Eg04a_MicroArchs/src_BSV/. Each shifts an 8-bit value x by a 3-bit value y.

Shifter_iterative.bsv	Sequential, iterative
Shifter_pipe_rigid.bsv	Pipelined. Rigid ("synchronous", assumes no gaps in data stream)
Shifter_pipe_elastic.bsv	Pipelined. Elastic ("asynchronous", accommodates gaps in input stream)

- The following are in Eg04b_MicroArchs/src_BSV/. They are generalizations of the previous three, such that each shifts an *n*-bit value x by a log(*n*)-bit value y. The testbench demonstrates instances where *n* = 16

Shifter_iterative.bsv	… ditto …
Shifter_pipe_rigid.bsv	… ditto …
Shifter_pipe_elastic.bsv	… ditto …

Building and running the codes

Each variation is built and run in the same way:

- In the "src_BSV" directory, create a symbolic link from "Shifter.bsv" to the variation of interest. E.g.,

```
% ln –s –f  Shifter_iterative.bsv  Shifter.bsv
```

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

```
% make  compile  link  simulate              // for Bluesim
% make  verilog  v_link  v_simulate          // for Verilog sim
```

Interface for the shifter(s)

All three variations of the shifter have the same interface (see file Shifter_IFC.bsv):

```
typedef Server #(Tuple2 #(Bit #(8), Bit #(3)),
                          Bit #(8))
        Shifter_IFC;
```

This is an example of a common BSV practice—to re-use “standard” interfaces already provided in the BSV library, rather than defining new, *ad hoc* interfaces for each new module:

```
interface Server #(t1, t2);
   interface Put #(t1) request;
   interface Get #(t2) response;
endinterface
```

(from the ClientServer library)

```
interface Put #(t1);
   method Action put (t1 x);
endinterface
```

```
interface Get #(t2);
   method ActionValue #(t2) get ();
endinterface
```

(from the GetPut library)

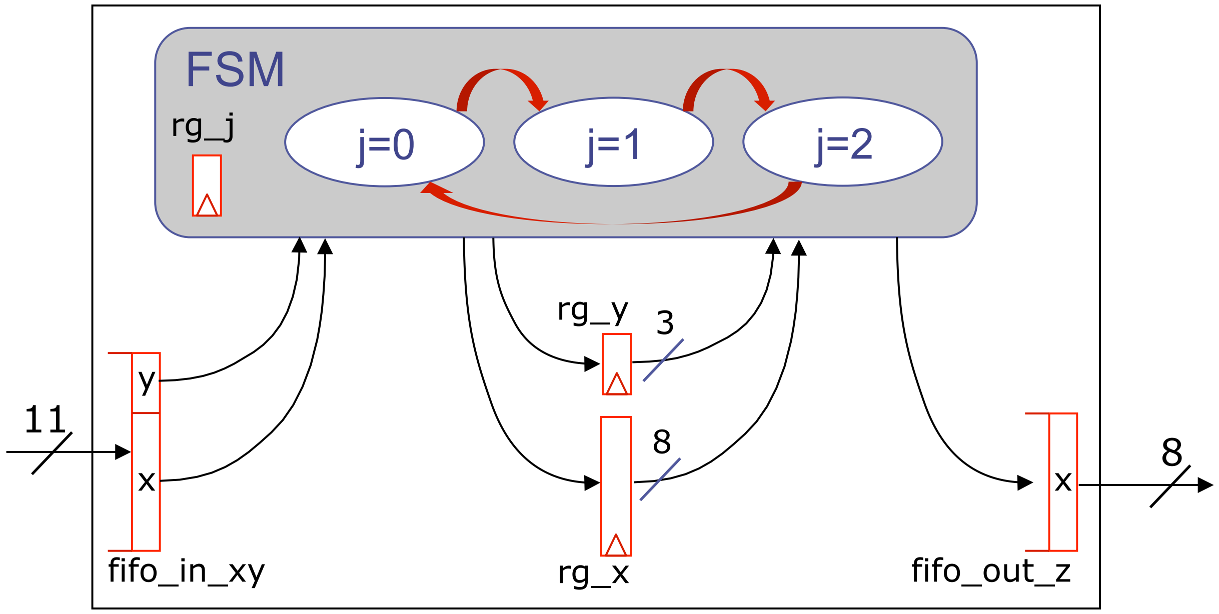
Note: these are similar to interfaces in the SystemC TLM 2.0 library

Our interface **`Shifter_IFC`** will therefore have

- a **`request.put`** method by which the environment can send a 2-tuple input
  - The 2-tuple is a pair of values, 8 bits (for x) and 3 bits (for y)
- a **`response.get`** method by which the environment can receive an 8 bit output

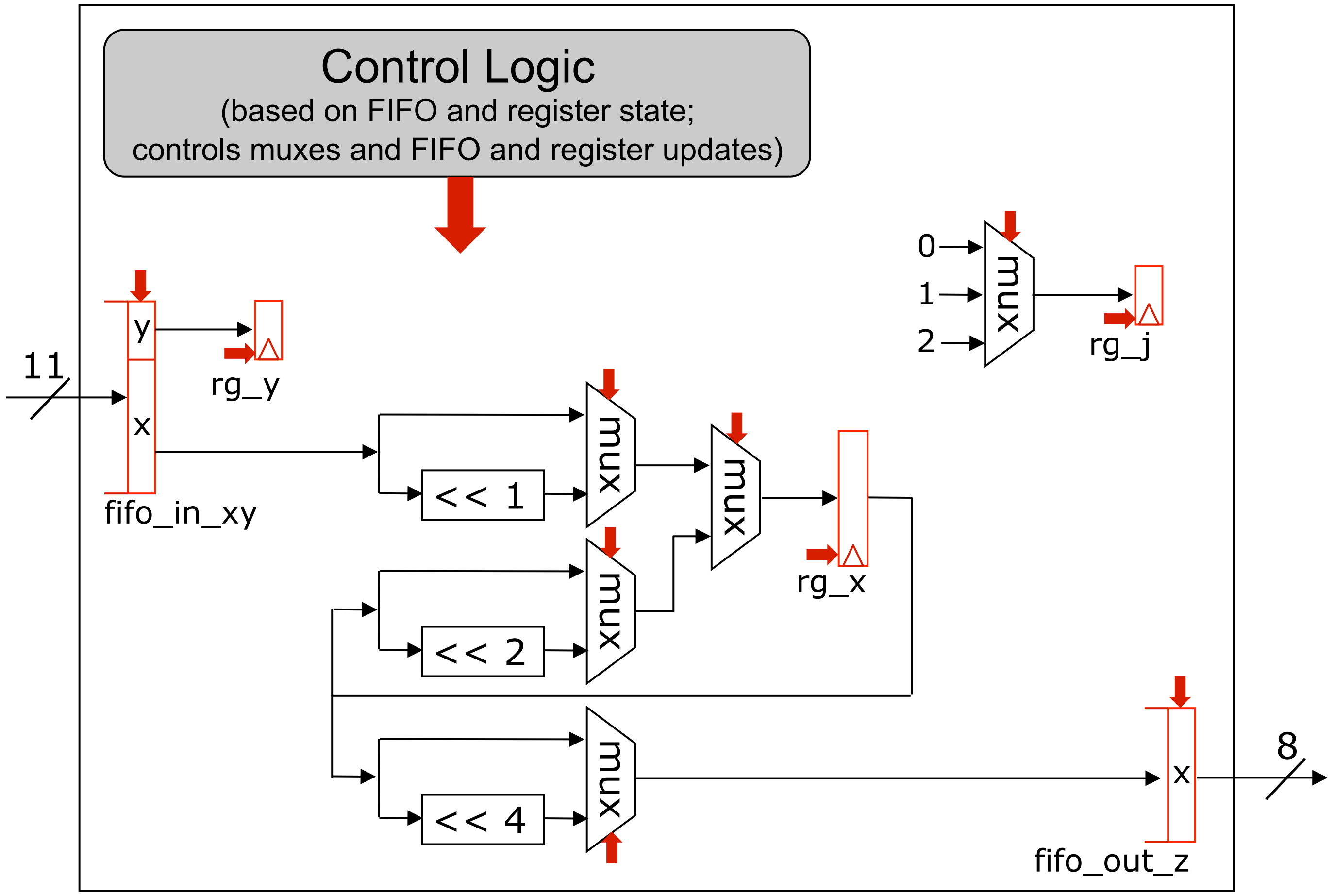
Sequential, iterative shifter

File: Eg04a_MicroArchs/src_BSV/Shifter_iterative.bsv



The FSM (and its actions) is implemented using 3 BSV rules

Sequential, iterative shifter: more detail



The Control Logic is automatically compiled by bsc from the BSV rules

A testbench to drive the shifter module

File: Eg04a_MicroArchs/src_BSV/Testbench.bsv

```
module mkTestbench (Empty);
   Shifter_Ifc  shifter <- mkShifter;

   Reg #(Bit #(4)) rg_y <- mkReg (0);

   rule rl_gen (rg_y < 8);
      shifter.request.put (tuple2 (8'h01, truncate (rg_y)));  // or rg_y[2:0]
      rg_y <= rg_y + 1;
   endrule

   rule rl_drain;
      let z <- shifter.get_z.get ();
      $display ("Output = %8b", z);
      if (z == 8'h80) $finish ();       // 8’b10000000
   endrule
endmodule: mkTestbench
```

rl_gen sends in the following inputs:

00000001   0
00000001   1
00000001   2
…
00000001   7

rl_drain should show the following outputs:

00000001
00000010
00000100
…
10000000

(The same testbench will be used for all three versions of the shifter)

Build and run, using the iterative shifter

- In the “src_BSV” directory, create a symbolic link from “Shifter.bsv” to the variation of interest:

```
% ln –s –f  Shifter_iterative.bsv  Shifter.bsv
```

- In the upper directory (Eg04a_MicroArchs or Eg04b_MicroArchs), build and run either using BDW or the ‘make’ commands, either with Bluesim or with Verilog sim, as described earlier

- Verify that the program produces the expected output

- The $displays on the input and output also print the clock cycle on which each input and output is done
  - Observe that after a start-up transient, input and output occur every 3 cycles. Why?

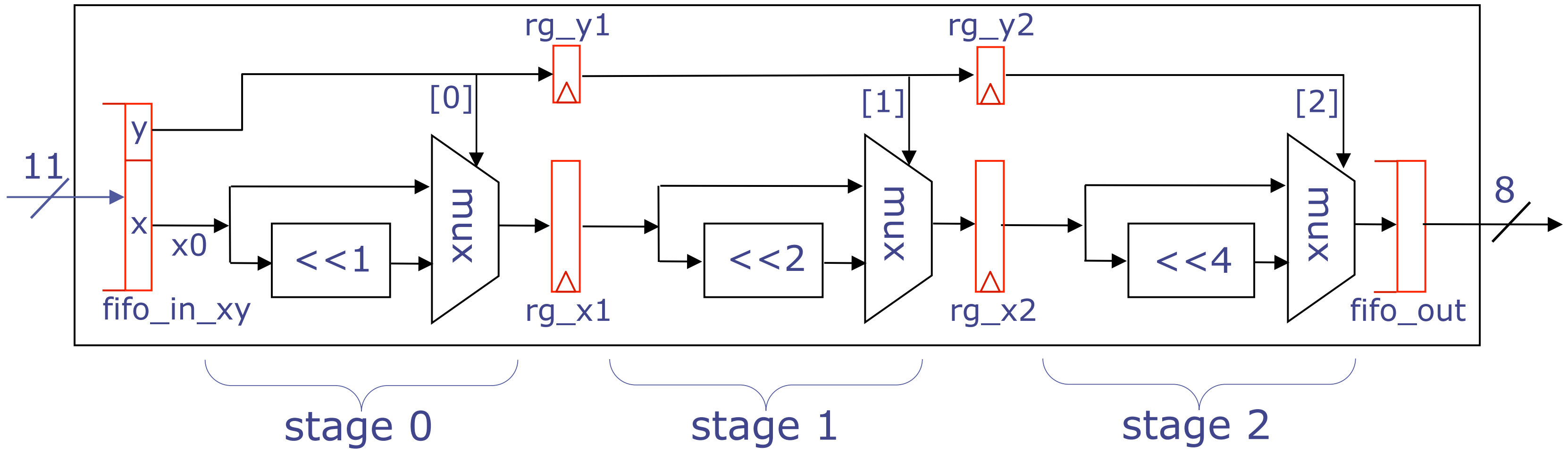
Time-out to reinforce some concepts

Please study the lecture: Lec_Interfaces_TLM
to understand the concepts Get, Put, Client and Server interfaces, and the mkConnection module.

Please also look at Section 10, “Pattern Matching” in the Reference Guide for more information on the “match” construct.

“Rigid” pipelined shifter

File: Eg02a_MicroArchs/src_BSV/Shifter_pipe_rigid.bsv



```
rule rl_all_together;
   // Stage 0
   match { .x0, .y0 } = fifo_in_xy.first; fifo_in_xy.deq;
   rg_x1 <= ((y0[0] == 0) ? x0 : (x0 << 1));
   rg_y1 <= y0;

   // Stage 1
   rg_x2 <= ((rg_y1[1] == 0) ? rg_x1 : (rg_x1 << 2));
   rg_y2 <= rg_y1;

   // Stage 2
   fifo_out_z.enq (((rg_y2[2] == 0) ? rg_x2 : (rg_x2 << 4)));
endrule
```

Build and run, using the rigid pipelined shifter

- In the “src_BSV” directory, create a symbolic link from “Shifter.bsv” to the variation of interest:

```
% ln –s –f  Shifter_pipe_rigid.bsv  Shifter.bsv
```

- In the upper directory (Eg04a_MicroArchs or Eg04b_MicroArchs), build and run either using BDW or the ‘make’ commands, either with Bluesim or with Verilog sim, as described earlier

- Verify that it is pipelined, i.e., that input and output happen on every clock
- However, the output does not seem to be fully correct:

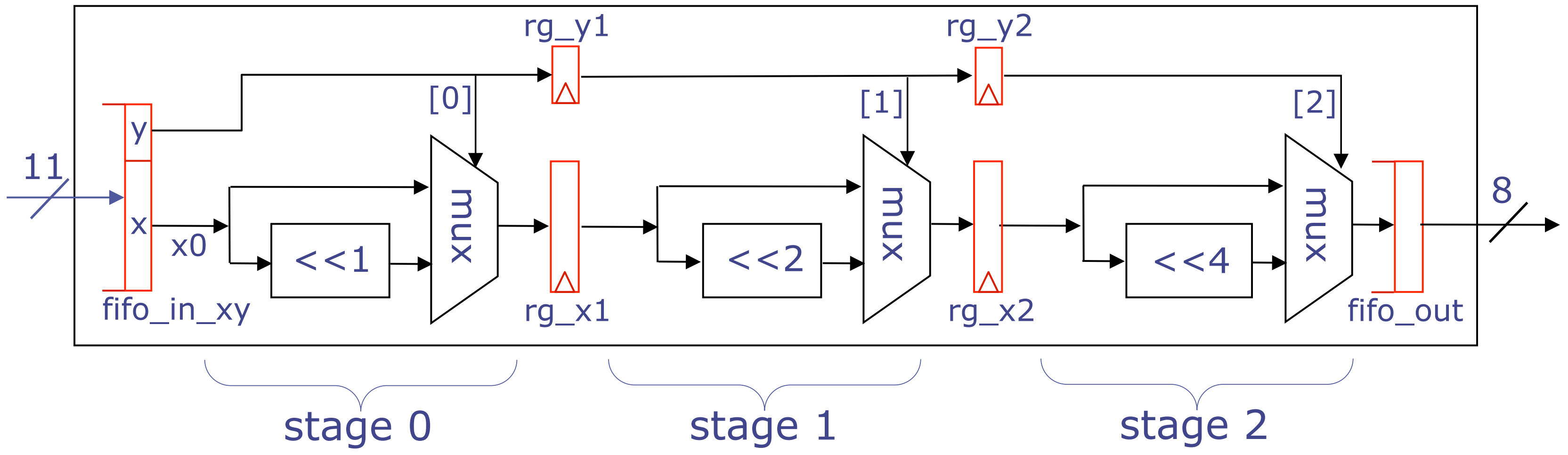
```
01010101
10101000
00000001
00000010
00000100
00001000
00010000
00100000
```

ok

Why?

… and then the program hangs

"Stranding" in the rigid shifter



Actually output is:

01010101
10101000
00000001
00000010
…
00010000
00100000

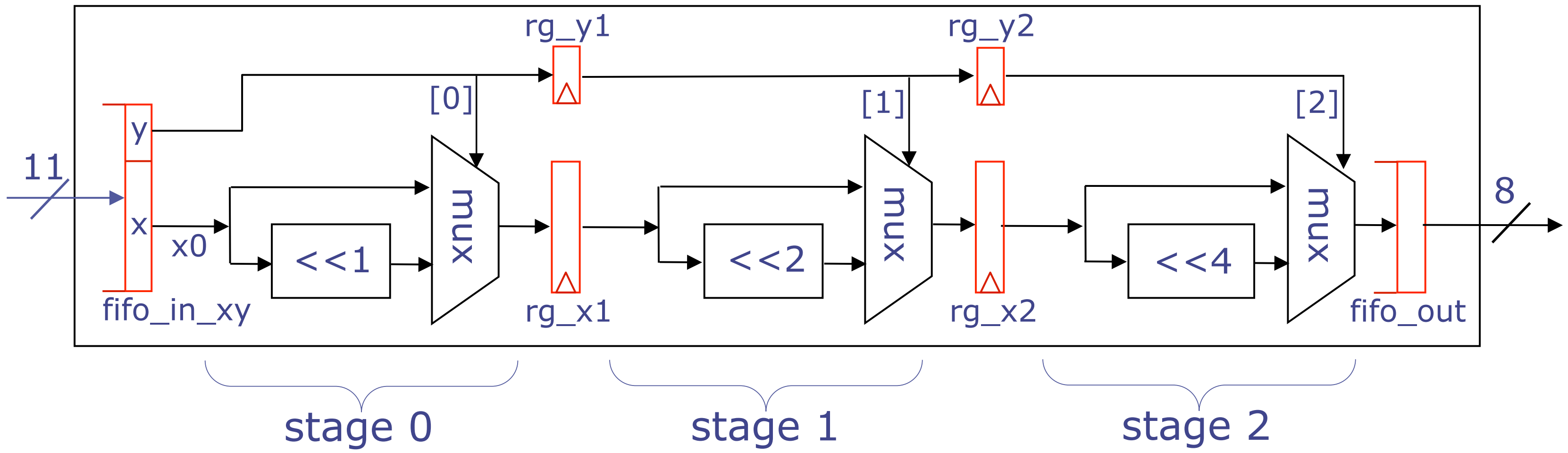
… and then the program hangs

The first two outputs are just based on the initial unspecified values in rg_x1, rg_y1, rg_x2 and rg_y2, as they are pushed through the pipeline.
bsc usually uses 'hA…A (10101010…1010) for initial values of unspecified state.

The remaining outputs are the correct outputs, but:

- when rl_gen stops feeding the input fifos,
- rl_all_together can no longer fire (since it invokes fifo_in_x.first whose method condition will be false)
- rl_drain can no longer fire if fifo_out is empty
- and so the last two values are "stranded" in rg_x1 and rg_x2

Observations about the rigid shifter

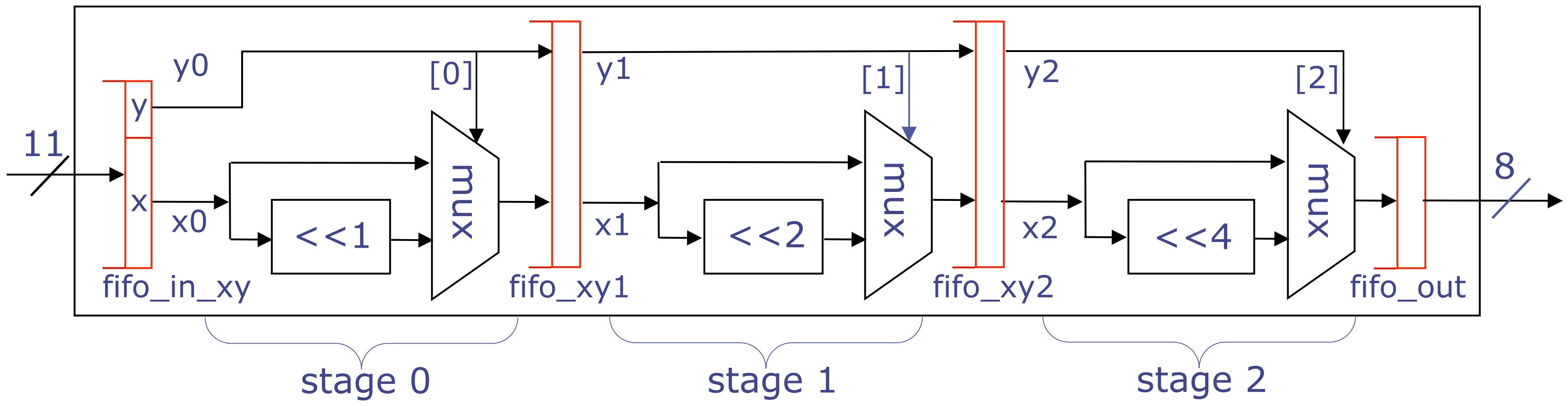


The “rigid” shifter is an example of a simple pipeline implementation that is ok if we have a non-stop, continuous stream of data with no gaps/bubbles. We also call such pipelines “synchronous”, or “lock-step”.

The key observation about the program structure is that all the actions in the pipe are expected to be simultaneous, and therefore placed in a single rule.

Elastic, pipelined shifter

File: Eg02a_MicroArchs/src_BSV/Shifter_pipe_elastic.bsv



```
module mkShifter (Shifter_Ifc);
   …
   FIFOF #(Tuple2 #(Bit #(8), Bit #(3))) fifo_xy1 <- mkFIFOF;
   FIFOF #(Tuple2 #(Bit #(8), Bit #(3))) fifo_xy2 <- mkFIFOF;

   rule rl_stage0;
      match { .x0, .y0 } = fifo_in_xy.first;  fifo_in_xy.deq;
      fifo_xy1.enq (tuple2 (((y0[0] == 0) ? x0 : (x0 << 1)), y0));
   endrule

   rule rl_stage1;
      match { .x1, .y1 } = fifo_xy1.first;  fifo_xy1.deq;
      fifo_xy2.enq (tuple2 (((y1[1] == 0) ? x1 : (x1 << 2)), y1));
   endrule

   rule rl_stage2;
      match { .x2, .y2 } = fifo_xy2.first;  fifo_xy2.deq;
      fifo_out_z.enq ((y2[2] == 0) ? x2 : (x2 << 4));
   endrule
   …
endmodule
```

We now have a separate rule for each stage.

Each rule can independently fire if its condition is true.

Build and run, using the elastic pipelined shifter

- In the "src_BSV" directory, create a symbolic link from "Shifter.bsv" to the variation of interest:

```
% ln –s –f  Shifter_pipe_elastic.bsv  Shifter.bsv
```

- In the upper directory (Eg04a_MicroArchs or Eg04b_MicroArchs), build and run either using BDW or the 'make' commands, either with Bluesim or with Verilog sim, as described earlier

- Verify that the program produces the expected output
- Verify that it is pipelined, i.e., that input and output happen on every clock

Summarizing what we’ve seen so far

- Iterative, rigid-pipelined, and elastic-pipelined structures are three examples of micro-architectural choices for the user.
  - They have different characteristics: area, clock speed, throughput, energy consumption, …
  - Which one is “best” depends on your design objectives.

- BSV does not (and should not!) make these choices for the user.
  - In creating software, algorithm design is best done by humans (not by programming languages).
  - Similarly, in creating hardware, architectural design is best done humans (not hardware design languages).
  - In both cases, languages can only facilitate quick and reliable expression of the choice made by the human designer, together with efficient implementation.

Generalizing the dynamic shifters for arbitrary bit-width

Please change directories
from Eg04a_MicroArchs/
to   Eg04b_MicroArchs/

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.

Generalized interface for the shifters

All three variations of the shifter have the same interface (see file Shifter_IFC.bsv):

```
typedef Server #(Tuple2 #(Bit #(n), Bit #(TLog #(n))),
                          Bit #(n))
        Shifter_IFC #(numeric type n);
```

The interface is now parameterized by ‘n’, the bit-width of the x input.

The bit-width of the y input is constrained to log(n), in order to express any shift amount from 0 to n.

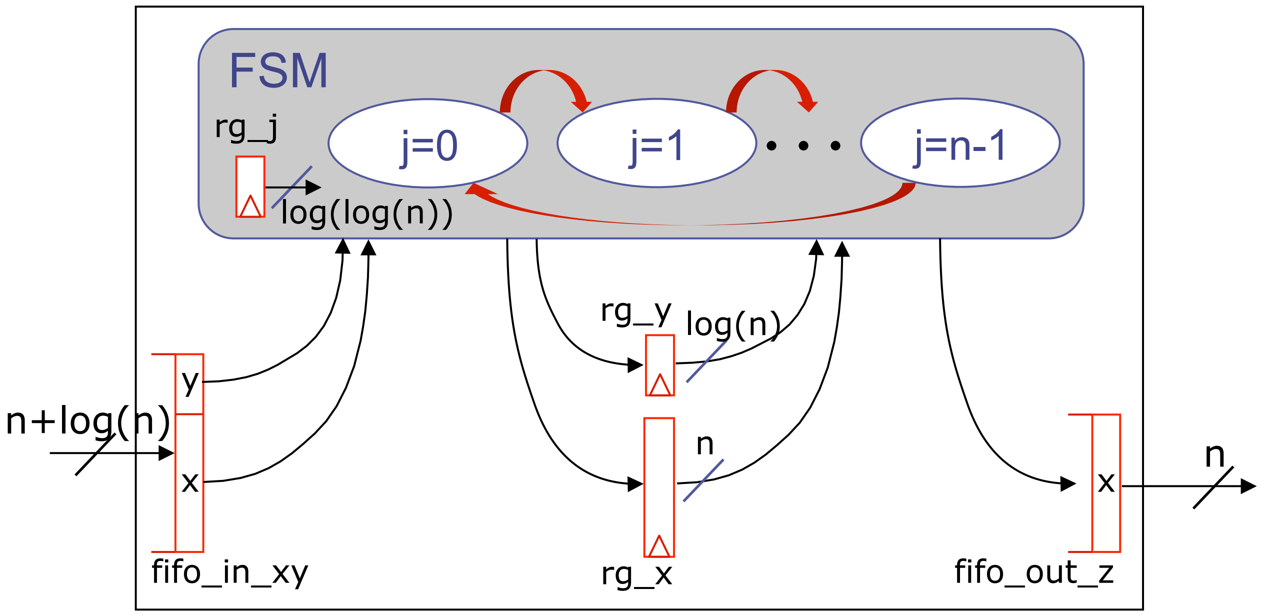
In BSV, certain types and type parameters can be *numeric types.*

Note that *numeric types* (which are only meaningful at compile time) are distinct from *numeric values* (which can of course occur at run time). This is why we use the special notation “TLog#(n)” to express a computation in numeric types.

Strictly speaking, TLog#(n) represents the ceiling of the log(n), i.e., an integer number of bits adequate to hold the binary representation of n.

Sequential, iterative n-bit shifter

File: Eg04b_MicroArchs/src_BSV/Shifter_iterative.bsv



- The FSM (and its actions) is implemented using n BSV rules
- The rules corresponding to j=1..(n-2) are generated in a for-loop
- rg_j is log(log(n)) bits wide, since it selects bits 0..log(n)-1 in y

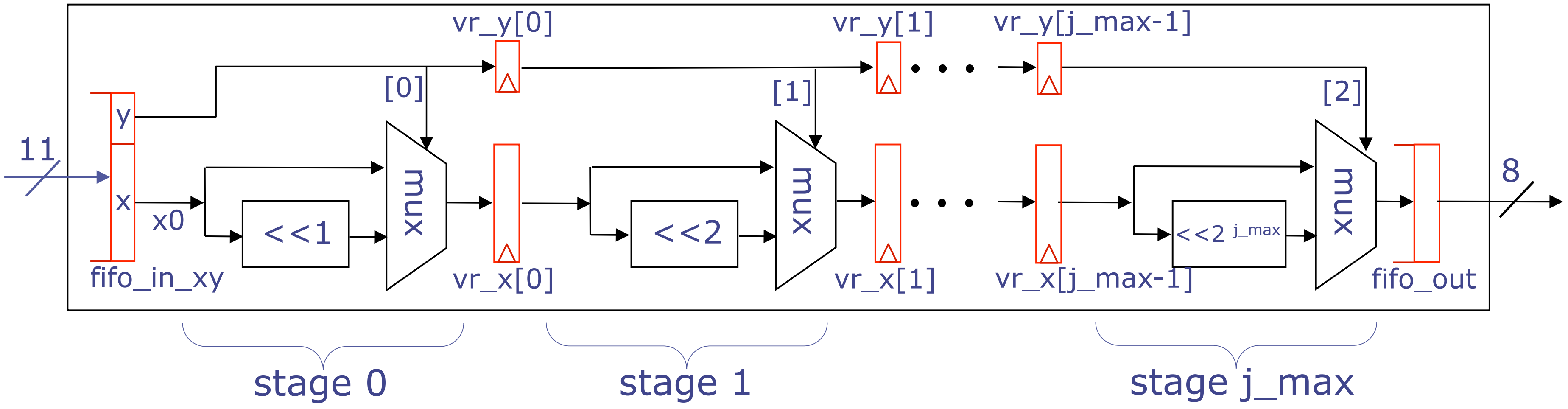
- Note: `rg_x << (2**j)` is not a dynamic shift: 2**j is compile-time constant

- BSV is very strict in type-checking—there are no automatic conversions
  The code uses these explicit conversions:

$$\begin{tikzcd} \boxed{\text{numeric type}} \arrow[r, "\text{valueOf()}"] & \boxed{\text{Integer}} \arrow[r, "\text{fromInteger()}"] & \boxed{\text{Bit \#(Tlog \#(Tlog \#(n)))}} \end{tikzcd}$$

“Rigid” pipelined n-bit shifter

File: Eg04b_MicroArchs/src_BSV/Shifter_pipe_rigid.bsv



- The x and y registers are generalized to *vectors* of log(n)-1 registers:

```
Vector #(TSub #(TLog #(n), 1), Reg #(Bit #(n)))         vr_x <- replicateM (mkRegU);
Vector #(TSub #(TLog #(n), 1), Reg #(Bit #(TLog #(n)))) vr_y <- replicateM (mkRegU);
```

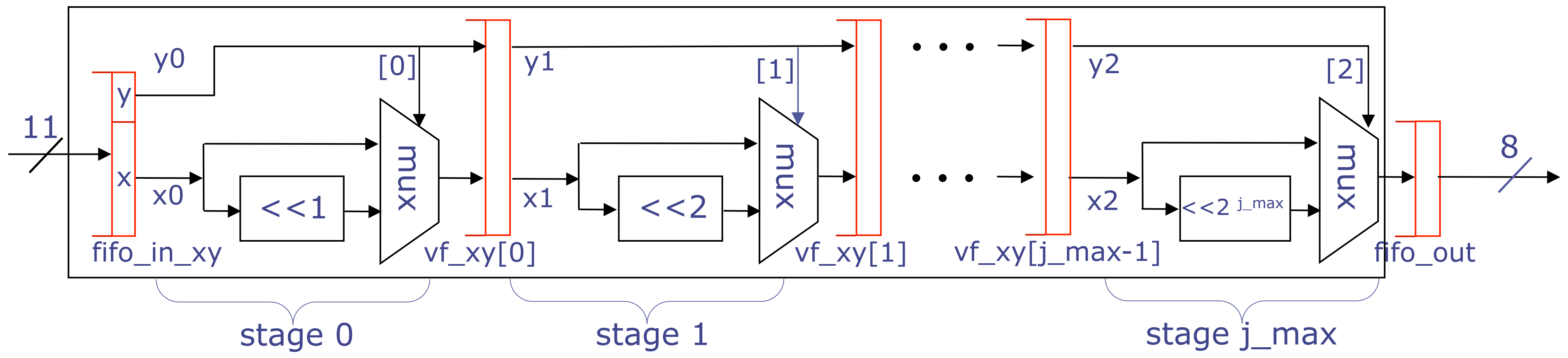
- The Actions of stages 1..j_max-1 are generated using a for-loop:

```
rule rl_all_together;
   // Stage 0
   …
   // Stage j: 1..j_max-1
   vr_x[j] <= ((vr_y[j-1][j] == 0) ? vr_x[j-1]: (vr_x[j-1] << (2**j)));
   vr_y[j] <= vr_y[j-1];

   // Stage j_max
   …
endrule
```

Elastic, pipelined n-bit shifter

File: Eg04b_MicroArchs/src_BSV/Shifter_pipe_elastic.bsv



- The xy FIFOs are generalized to *vectors* of log(n)-1 FIFOs:

```
Vector #(TSub #(TLog #(n), 1),
         FIFOF #(Tuple2 #(Bit #(n),
                          Bit #(TLog #(n))))) vf_xy <- replicateM (mkFIFOF);
```

- The Rules for stages 1..j_max-1 are generated using a for-loop:

```
for (Integer j = 1; j < j_max; j = j + 1)
   rule rl_j;
      match { .x1, .y1 } = vf_xy[j-1].first;  vf_xy[j-1].deq;
      vf_xy[j].enq (tuple2 (((y1[j] == 0) ? x1 : (x1 << (2**j))), y1));
   endrule
```

Synthesis hierarchy

- In the original 8-bit shifters (in Eg04a_MicroArchs/ directory), in front of each ‘module’ line, there is a (*synthesize*) attribute:

```
(* synthesize *)
module mkShifter (Shifter_IFC);
 …
endmodule
```

- When generating Verilog, this creates a ‘mkShifter.v’ file with a ‘mkShifter’ Verilog module

- In the generalized n-bit shifters (in Eg04b_MicroArchs/ directory), these (*synthesize*) attribute are removed.
- This is because BSV cannot separately synthesize *polymorphic* modules; they can only be inlined into a parent module
- Instead, in Testbench.bsv, we have created a specific instance of the module (for shifting 16-bit values); since this is no longer polymorphic, it can be separately synthesized. This is the module actually instantiated in the module mkTestbench:

```
(* synthesize *)
module mkShifter_16_4 (Shifter_IFC #(16));
   let m <- mkShifter;
   return m;
endmodule
```

- The above is a common idiom in BSV code, for creating a separately synthesized instance of a polymorphic module

Build and run the generalized shifters

- Build and run the generalized shifters (in Eg04b_MicroArchs/ directory) in a similar manner to how you ran the 8-bit shifters (in Eg04a_MicroArchs/ directory)
- Verify that the behaviors are as expected (including the throughputs)

Suggested exercises

- Change the program to *rotate* x by n bits, instead of shifting, i.e., instead of losing bits at the MSB end and shifting in zeroes at the LSB end, the MSB bits should be shifted in at the LSB end.

- Change any one of the mkShifter modules so that it takes a *static* boolean parameter such that:
  - If True, we get a circuit that performs left-shifting.
  - If False, we get a circuit that performs left-rotation.
  - (Note: this is a fixed-function circuit, either left-shifting or left-rotation, chosen at compile time.)

- Change any one of the *pipelined* mkShifter modules so that it can perform left- and right-shifting and left- and right-rotation, selected dynamically.
  - Change the interface so that the input is now a 3-tuple, where the new, third component is an "opcode" specifying left/right shift/rotate. Define an enum type for this opcode. Note: successive 3-tuple inputs may carry different opcodes, i.e., different pipeline stages may be performing different operations at the same time.
  - (Note: this circuit is a piece of a full-blown "ALU" for a CPU.)

Summary

- These variations on a “dynamic shifter” illustrate how BSV can be used to express a range of micro-architectures, from FSMs to rigid/synchronous pipelines to elastic/ asynchronous pipelines
- They also show how architectures can be parameterized flexibly

End