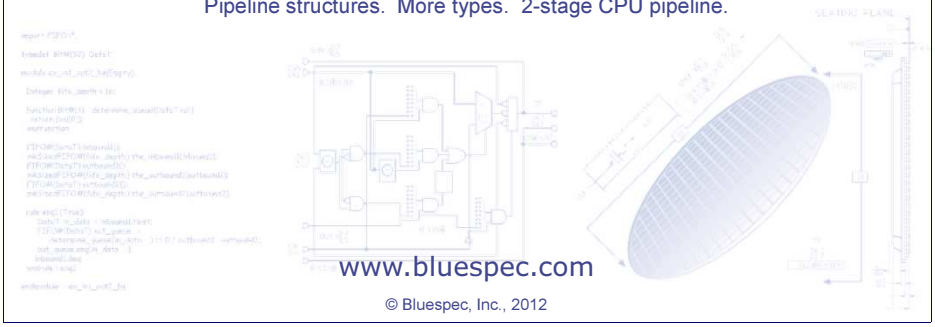
bluespec

BSV Training

Section: Examples

Pipeline structures. More types. 2-stage CPU pipeline.



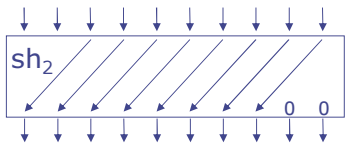
www.bluespec.com

Example: a pipelined shifter

- Goal: circuit to implement a left-shift by a dynamic amount *s* : z = shift (x, y)
  i.e., z = x left-shifted by y positions
- Suppose x has type Bit #(8) and y has type Bit #(3)
- Strategy (algorithm):
  - Shift x by y =

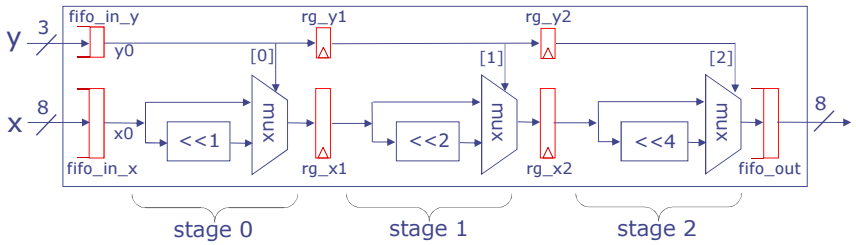
    shift x by 4 (= 2^2) if y[2] == 1
    and x by 2 (= 2^1) if y[1] == 1
    and x by 1 (= 2^0) if y[0] == 1

  - Note: shifting by 2^j is trivial: it's just a "lane change" involving only wires, no circuit logic:



bluespec

A classical synchronous, lock-step, pipelined shifter




```
   rule rl_all_together;
      // Stage 0
      let x0 = fifo_in_x.first; fifo_in_x.deq;
      let y0 = fifo_in_y.first; fifo_in_y.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
```


bluespec

Synchronous shifter module containing the rule

```
interface Shifter_Ifc;
   interface Put #(Bit #(8)) put_x;
   interface Put #(Bit #(3)) put_y;
   interface Get #(Bit #(8)) get_z;
endinterface
```

Rather than inventing new methods, we often re-use library interfaces like Get/Put (see below) that capture common design patterns.

```
module mkShifter (Shifter_Ifc);
   FIFOF #(Bit #(8))  fifo_in_x  <- mkFIFOF;
   FIFOF #(Bit #(3))  fifo_in_y  <- mkFIFOF;
   FIFOF #(Bit #(8))  fifo_out_z <- mkFIFOF;

   Reg #(Bit #(8)) rg_x1 <- mkRegU;
   Reg #(Bit #(3)) rg_y1 <- mkRegU;

   Reg #(Bit #(8)) rg_x2 <- mkRegU;
   Reg #(Bit #(3)) rg_y2 <- mkRegU;

   rule rl_all_together;
      …
      … shown on previous slide …
      …
   endrule

   interface put_x = toPut (fifo_in_x);
   interface put_y = toPut (fifo_in_y);
   interface get_z = toGet (fifo_out_z);
endmodule
```

Some BSV library interfaces

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

interface Get #(t);
   method ActionValue #(t) get
();
endinterface

interface FIFOF #(t);
   method Action enq (t x);
   method t first ();
   method Action deq ();
   method Bool notFull;
   method Bool notEmpty;
   method Action clear ();
endinterface
```

These are BSV library functions that convert a FIFOF interface into a Get or Put interface (each is just a few lines of BSV)

bluespec

A testbench to drive the shifter module

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

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

   rule rl_gen (rg_y < 8);
      shifter.put_x.put (8'h01);
      shifter.put_y.put (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: mkTest
```

rl_gen sends in the following inputs:

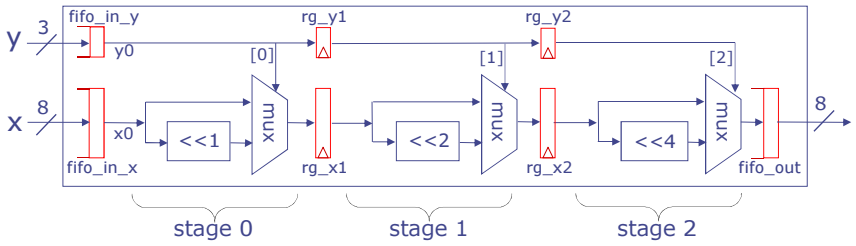
00000001 0
00000001 1
00000001 2
…
00000001 7

rl_drain should see the following outputs:

00000001
00000010
00000100
…
10000000

bluespec

Actual output with the synchronous shifter



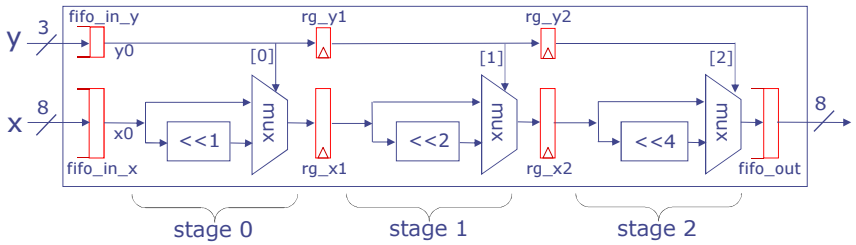
But what we actually see is:

01010101
10101000
00000001
00000010
…
00010000
00100000
… and then the program hangs

Why?

bluespec

"Stranding" in the synchronous "lock-step" shifter



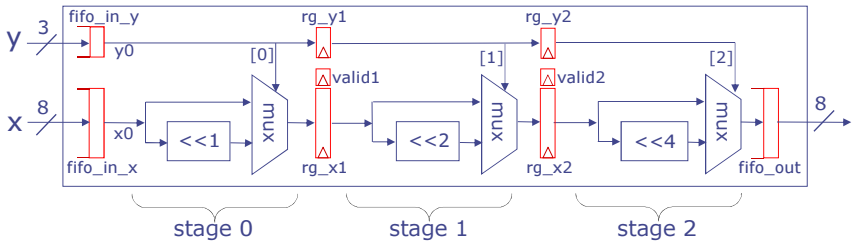
But what we actually see is:
01010101
10101000
00000001
00000010
…
00010000
00100000
… and then the program hangs

The first two lines 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 lines are the right 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

bluespec

A more flexible synchronous pipeline



We can solve both problems (garbage initial values, and stranding) by adding "valid" bits to the intermediate registers, and extending the rule to check
- when the input fifos are empty,
- when rg_x1 is valid
- when rg_x2 is valid

and adjusting its behavior accordingly

bluespec

A small sidebar on the "Maybe" type

BSV (and SystemVerilog) have kind of type called "tagged unions".
One tagged union type frequently used in BSV is the "Maybe" type.
(Reference Manual section B.2.10)

The type declaration

```
typedef union tagged {
    void  Invalid;
    t     Valid;
} Maybe #(type t)
  deriving (Eq, Bits);
```

A "Maybe#(t)" value is
- *either "invalid" (with no associated value, i.e., void)*
- *or "valid" with an associated value of type "t"*

i.e., a "valid" bit along with a value

Creating values of this type

```
tagged Invalid
```
creates 0 (invalid) along with a don't care value

```
tagged Valid expression
```
creates 1 (valid) along with the value of the expression

Using values of this type, with "pattern matching"

```
if (value matches tagged Valid .x)
   … here you can use x, the valid associated value …
else
   … here you handle the "invalid" case …
```

Tagged unions are similar to "unions" in C/C++, except that tagged unions are type-safe, whereas unions are not. There is no way to examine the value when the valid bit is "invalid".

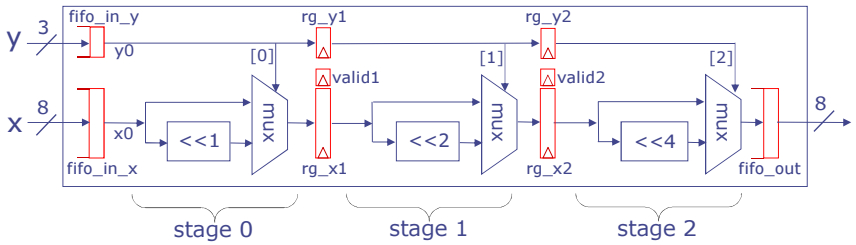
bluespec

A more flexible synchronous pipeline

```
module mkShifter (Shifter_Ifc);
   …
   Reg #(Maybe #(Bit #(8))) rg_x1 <- mkReg (tagged Invalid);
   Reg #(Bit #(3))          rg_y1 <- mkRegU;

   Reg #(Maybe #(Bit #(8))) rg_x2 <- mkReg (tagged Invalid);
   Reg #(Bit #(3))          rg_y2 <- mkRegU;

   rule rl_all_together;
      // Stage 0
      Bit #(3) y0 = ?;
      if (fifo_in_x.notEmpty) begin
 let x0 = fifo_in_x.first; fifo_in_x.deq;
 y0 = fifo_in_y.first; fifo_in_y.deq;
 rg_x1 <= tagged Valid ((y0 [0] == 0) ? x0 : (x0 << 1));
      end
      else
 rg_x1 <= tagged Invalid;
      rg_y1 <= y0;

      // Stage 1
      if (rg_x1 matches tagged Valid .x1)
 rg_x2 <= tagged Valid ((rg_y1 [1] == 0) ? x1 : (x1 << 2));
      else
 rg_x2 <= tagged Invalid;
      rg_y2 <= rg_y1;

      // Stage 2
      if (rg_x2 matches tagged Valid .x2)
 fifo_out_z.enq (((rg_y2 [2] == 0) ? x2 : (x2 << 4)));
   endrule
   …
endmodule
```


bluespec

A more flexible synchronous pipeline



When we build and simulate this code (with the same testbench, we see the expected outputs, and simulate terminates normally.

rl_gen sends in the following inputs:

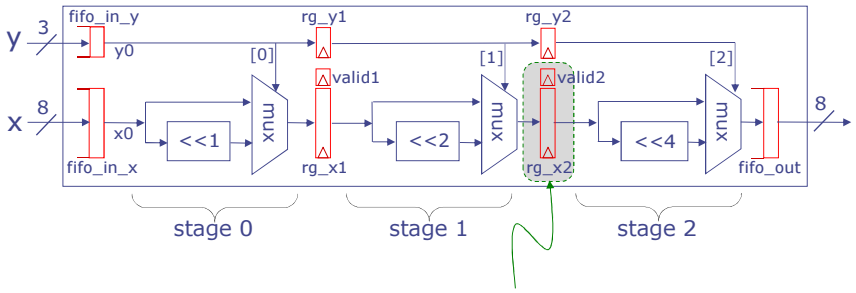
```
00000001  0
00000001  1
00000001  2
…
00000001  7
```

rl_drain should see the following outputs:

```
00000001
00000010
00000100
…
10000000
```


bluespec

Valid bits and FIFOs



A register with a valid bit can be viewed as a 1-element FIFO:

- When "valid" is true, the FIFO is "full"
- When "valid" is false, the FIFO is "empty"
- Storing a value and setting "valid" is "enqueing" an item on the FIFO
- Removing a value and resetting "valid" is "dequeuing" an item from the FIFO

With this observation, the flexible pipeline can be written more succinctly and elegantly with FIFOs than with explicit valid bits

- Much of the explicit control logic ("is fifo_in non-empty?" "is the valid bit true?") disappears into (is handled by) the method conditions on the FIFOs

bluespec

A small sidebar on "Tuple" types

A 2-tuple is just a pair of values; a 3-tuple is a triple, … and so on

The 2-tuple type:

`Tuple2 #(t1, t2)` — *A pair of values, the first one of type t1, and the second of type t2*

Creating values of this type

`tuple2 (expression1, expression2)` — *creates a value with two components, the value of expression1 and the value of expression2*

Using values of this type: functions to extract components:

`tpl_1 (expression), tpl_2 (e), …` — *extract the j'th component of tuple that is value of expression*

Using values of this type, with "pattern matching"

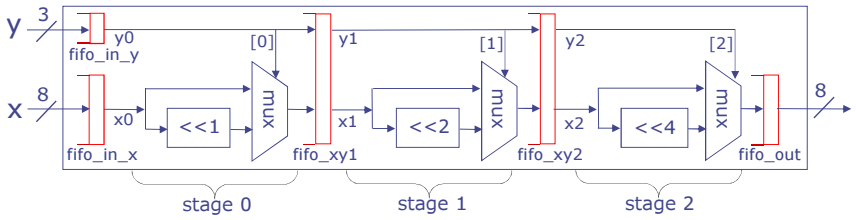
`match { .x, .y } = expression;` — *declares new variables x and y, and binds them to the components of the 2-tuple value of expression*

2-tuples are just structs with 2 fields (in general, an *n*-tuple is a struct with *n* fields).

But tuples are so useful and common that they're pre-defined in BSV.

bluespec

Elastic, pipelined shifter



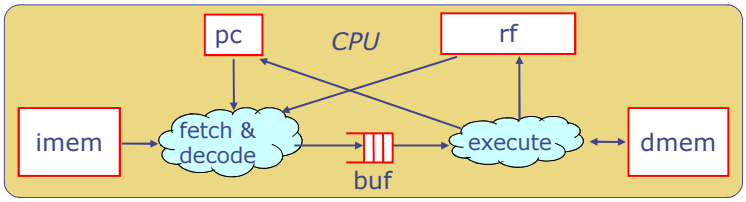

```
module mkShifter (Shifter_Ifc);
   …
   FIFOF #(Tuple2 #(Bit #(8), Bit #(3))) fifo_xy1 <- mkFIFOF;
   FIFOF #(Tuple2 #(Bit #(8), Bit #(3))) fifo_xy2 <- mkFIFOF;

   rule rl_stage0;
      let x0 = fifo_in_x.first;  fifo_in_x.deq;
      let y0 = fifo_in_y.first;  fifo_in_y.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
   …
```

We now have a separate rule for each stage.

Each rule can independently fire if its condition is true.

bluespec

Small example: simple 2-staged pipeline processor




```
module mkCPU (Empty);

   Reg#(Addr)  pc    <- mkReg (0);
   RegFile     rf    <- mkRegFile;     // register file
   IMem        imem  <- mkIMem;        // instruction mem
   DMem        dmem  <- mkDMem;        // data mem
   SFIFO       buf   <- mkSFIFO;       // searchable FIFO

   … rules for behavior … // (in following slides)

endmodule: mkCPU
```

(assume we're given these sub-modules as primitives or pre-defined)

bluespec

Types for instructions and decoded instructions

```
typedef  Bit #(32) Word;
typedef  Bit #(32) Addr;
typedef  Bit #(32) Data;

typedef  Bit#(4) RegName;
```

Simple typedefs (left) are just synonyms for readability; all these types are equivalent.

Enum and struct typedefs (below) define new types, not equivalent with any other type. (So, type-checking prevents misuse.)

```
typdef enum { Add, Bz, Ld, St } Opcode deriving (Bits, Eq);
```

```
typedef struct {
   Opcode   op;
   RegName  dest;
   RegName  src1;
   RegName  src2;
} Instr
  deriving (Bits);
```

```
typedef struct {
   Opcode    op;
   RegName   dest;
   Bit #(32) v1;
   Bit #(32) v2;
} DecodedInstr
  deriving (Bits);
```

"deriving (Eq)" tells bsc to pick a "natural" equality comparison operator for this type.

"deriving (Bits)" tells bsc to pick a "natural" bit-representation for this type.

(detailed treatment in a later section)

bluespec

Sub-modules of the processor, and their interfaces

The interface of the searchable FIFO (buf) between the decode and exec stages

```
interface SFIFO;
   method Bool notStall (Instr  instr);
    // True if FIFO is not full and does not contain
    // a decoded instr whose dest is
    // instr.src1 or instr.src2
    // ("instruction dependency")

   method Action enq (DecodedInstr di);
    // Enqueue decoded instr into FIFO

   method DecodedInstr  first;    // return 1st element
   method Action deq;             // discard 1st element

   method Action clear;           // empty the FIFO
endinterface
```


bluespec

Sub-modules of the processor, and their interfaces

The interface of the register file (rf)

```
interface RegFile;
   method Word sel1 (RegName  r);         // 1st read-
port

   method Word sel2 (RegName  r);         // 2nd read-
port

   method Action upd (RegName r, Word v); // write-port
endinterface
```

The interface of the instruction memory (imem)

```
interface IMem;
   method Word load (Addr  a);    // read-port
endinterface
```

The interace of the data memory (dmem)

```
interface DMem;
   method Word load (Addr  a);           // read-port
   method Action upd (Addr a, Word v);   // write-port
endinterface
```


bluespec

The processor's fetch-and-decode rule

```
Instr  instr = imem.load (pc); // the current instruction

rule fetch (buf.notStall (instr));
   let  di = DecodedInstr { op:   instr.op;
                            dest: instr.dest;
                            v1:   rf.sel1 (instr.src1);
                            v2:   rf.sel2 (instr.src2); };
   buf.enq (di);
   pc <= pc + 1;
endrule: fetch
```

- The rule will only execute if 'buf.notStall(instr)' is True
  - (FIFO not full, and no instruction dependency)
- 'let di = ...' builds a decoded instruction struct value from the fields in 'instr' and value read from the register file
- When the rule executes, it has two simultaneous actions: enqueue di and increment the PC
- Note: blindly fetches next instr, even if current instr is a Bz instruction!

bluespec

The processor's execute rule

```
rule exec;
   let  di = buf.first; // decoded instruction
   case (di.op)
      Add: begin rf.upd (di.dest, di.v1 + di.v2);
                 buf.deq; end
      Ld:  begin rf.upd (di.dest, dmem.load (di.v1));
                 buf.deq;  end
      St:  begin dmem.store (di.v1, di.v2);
                 buf.deq;  end
      Bz:  if (di.v1 == 0) begin  pc <= di.v2;
                                  buf.clear ();  end
           else buf.deq;
   endcase
endrule: exec
```

- The rule will not execute if the FIFO buf is empty
  - (implicit condition of 'first' and 'deq' methods)
- The action of the rule depends on the branch of the case statement taken

bluespec

Rule non-determinism in this example

- What if both rules are enabled?
  - We can choose either rule—*it does not matter for functional correctness*!
    - If we repeatedly choose the exec rule, eventually it will become disabled due to the FIFO becoming empty, and then we'll have to choose the fetch rule
    - If we repeatedly choose the fetch rule, eventually it will become disabled due to the FIFO becoming full or because there's an instruction dependency, and then we'll have to choose the exec rule
- In this example, the non-determinism does not matter
  - (but in other examples, non-determinism may matter)
  - Formal verification/specification people like it this way—don't *overspecify* when you don't have to

bluespec

Rule ordering helps reasoning about correctness

- Shared resource contention:
  - Fetch rule increments PC, enqueues into FIFO
  - Exec rule, for a Bz instruction, updates PC to branch target, clears the FIFO
- Is there a potential race condition? E.g.,
  - PC gets incremented, but FIFO is cleared?
  - Mispredicted instruction enq'd in FIFO, but PC updated to branch target?
- Rule ordering guarantees that this cannot happen!
  - Logically, either the fetch rule executes before the exec rule, or *vice versa*
    - *(but note that they can result in different performance)*
  - We can reason that we have a correct state either way
- Rule-at-a-time semantics helps us reason about correctness
  - We'll discuss performance in a later section

bluespec

Expressive power of BSV

These examples give a flavor of the expressive power of BSV:

- In BSV you can express any micro-architecture or architecture
- At one extreme, a special case, are classical, Verilog-like, synchronous, lock-step designs
- But you can also do elastic, automatically flow-controlled designs with complex concurrency. Rule and method conditions naturally lead to such designs.
- The former are historically more familiar (RTL); but the latter:
  - Are more scalable (large, globally synchronous designs are very fragile, rigid, difficult to change, difficult to plug-and-play, can have clock and reset distribution problems, etc.)
  - Are more consistent with modern GALS philosophy (Globally Asynchronous, Locally Synchronous)
- Independently, whether your design is synchronous or asynchronous, BSV also gives you much richer types and stronger type-checking than RTL

bluespec

bluespec

End

Questions?
Join online forums at www.bluespec.com, and ask your question,
or send an e-mail to support@bluespec.com