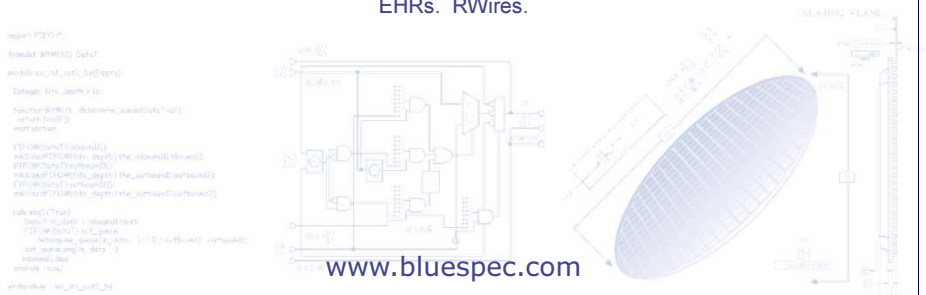
bluespec

BSV Training

Section: Enabling greater concurrency (packing more rules into clocks)

EHRs. RWires.

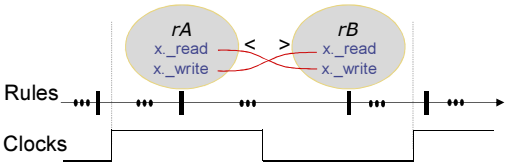


www.bluespec.com

Enabling greater concurrency

- We have seen that two rules rA and rB can fire in the same clock only if the ordering constraints between methods in rA and methods in rB are satisfied.
- One way for a rule to communicate with another is via the register primitive. Two rules that read/write a register can never be scheduled in the same clock:

```
rule rA;
   x <= x + 1;
endrule

rule rB;
   x <= x * 2;
endrule
```



- For greater concurrency in rules, we need another primitive whose method ordering constraints allow this.
- In HW, of course, we communicate within clocks using wires.
- In BSV, we use a primitive called *EHRs*, which is an abstraction of wires that fits nicely into rule semantics.

bluespec

A motivating example

- Suppose we want to build a two-port, saturating, up/down counter of 4-bit signed integer values, with the following interface:

```
interface UpDownSatCounter_Ifc;
   method ActionValue #(Int #(4)) countA (Int #(4) delta);
   method ActionValue #(Int #(4)) countB (Int #(4) delta);
endinterface
```

- The “two ports” are the two methods countA and countB
- A module implementing this interface has internal state holding the current value of the counter (Int #(4) type, so range is -8 to +7)
- When either method is called,
  - The internal state is incremented by delta (range: -8 to +7), but saturates at +7 on overflow and at -8 on underflow
  - The old value of the counter is returned as the result of the method

Note: because of finite precision and saturation, “count” operations are not commutative, as in conventional arithmetic; order matters!

bluespec

An implementation using ordinary registers (v1)

```
module mkUpDownSatCounter (UpDownSatCounter_Ifc);
   Reg #(Int #(4)) ctr <- mkReg (0);

   function ActionValue #(Int #(4)) fn_count (Int #(4)
delta);
      actionvalue
         // Extend the precision to avoid over/under flows
         Int #(5) new_val = extend (ctr) + extend (delta);
         if (new_val > 7) ctr <= 7;
         else if (new_val < -8) ctr <= -8;
         else ctr <= truncate (new_val);

         return ctr;   // note: returns old value
      endactionvalue
   endfunction

   method countA (Int #(4) deltaA) = fn_count (deltaA);
   method countB (Int #(4) deltaB) = fn_count (deltaB);
endmodule
```

Since both methods do the same thing, we abstract out their common behavior into a function fn_count()

BSV note:

- “extend (e)” sign-extends for Int#(n), and zero-extends for Bit#(n) and UInt#(n)
- “truncate (e)” drops MSBs, taking care of sign bits etc.
- The number of bits extended/truncated depends on the input and output type widths

bluespec

A testbench to drive the up/down counter module

```
module mkTest (Empty);
   UpDownSatCounter_Ifc  ctr <- mkUpDownSatCounter;
   Reg #(int) step <- mkReg (0);
   Reg #(Bool) flag0 <- mkReg (False); Reg #(Bool) flag1 <- mkReg (False);

   function Action count_show (Integer rulenum, Bool a_not_b, Int #(4) delta);
      action
          let x <- (a_not_b ? ctr.countA (delta) : ctr.countB (delta));
          $display ("cycle %0d, r%0d: is %0d, count (%0d)", cur_cycle, rulenum, x, delta);
      endaction
   endfunction

   // Rules 0-9 are seqential, just testing one method at a time
   rule r0  (step == 0);  count_show (0, True,   3); step <= 1;  endrule
   rule r1  (step == 1);  count_show (1, True,   3); step <= 2;  endrule
                    ... and similarly, sequentially feed deltas of  3,3, -6,-6,-6,-6, 7, 3,
   // Concurrent execution
   rule r10 (step == 10 && !flag0); count_show (10,True,   6); flag0 <= True; endrule
   rule r11 (step == 10 && !flag1); count_show (11,False, -3); flag1 <= True; endrule

   // Show final value
   rule r12 (step == 10 && flag0 && flag1); count_show (12,True, 0); $finish; endrule
endmodule: mkTest
```

In rules 0-9, we call either countA or countB with deltas: 3,3,3,3, -6,-6,-6,-6, 7, 3
The rule conditions and step assignments force them to fire 1 rule per clock (and so it doesn't matter whether we call countA or countB in these rules).

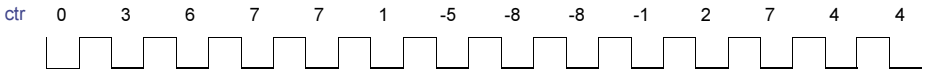
Rules 10 and 11 could potentially fire concurrently (if scheduling permits).

Rule 12 just displays the final counter value and exits.

bluespec

Expected behavior and outputs for v1

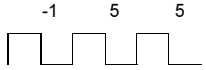
We expect the following behavior if r10 fires in the clock before r11:



rule	r0	r1	r2	r3	r4	r5	r6	r7	r8	r9	r10	r11	r12
delta	3	3	3	3	-6	-6	-6	-6	7	3	6	-3	0
outputs	0	3	6	7	7	1	-5	-8	-8	-1	2	7	4

We expect the following behavior if r11 fires in the clock before r10:

• • • *same for r0 through r9* • • •



r11	r10	r12
-3	6	0
2	-1	5

bluespec

Actual output for v1

When we compile the program (v1), *bsc* produces the following message:

```
Warning: "Test.bsv", line 16, column 8: (G0010)
  Rule "r10" was treated as more urgent than "r11". Conflicts:
    "r10" cannot fire before "r11": calls to ctr.countA vs. ctr.countB
    "r11" cannot fire before "r10": calls to ctr.countB vs. ctr.countA
```

This is saying:

- r10 and r11 conflict; they cannot be scheduled in the same clock
  (because countA and countB conflict, because they both read and write the "ctr" register inside mkUpDownSatCounter)
- *bsc* has chosen to give priority to r10, i.e., if both r10 and r11 are enabled in the same clock, the scheduling logic will allow r10 to fire and will suppress r11
  (r11 could fire, and indeed it does, in the next clock, when r10 is no longer enabled)
- You can force *bsc* to give the opposite priority by adding a "descending_urgency" attribute to the module

When we run the program (v1), we see:

(per first schedule in previous slide)

```
cycle 1, r0: is 0, count (3)
cycle 2, r1: is 3, count (3)
cycle 3, r2: is 6, count (3)
cycle 4, r3: is 7, count (3)
cycle 5, r4: is 7, count (-6)
cycle 6, r5: is 1, count (-6)
cycle 7, r6: is -5, count (-6)
cycle 8, r7: is -8, count (-6)
cycle 9, r8: is -8, count (7)
cycle 10, r9: is -1, count (3)
cycle 11, r10: is 2, count (6)
cycle 12, r11: is 7, count (-3)
cycle 13, r12: is 4, count (0)
```


bluespec

v1 is not really a "2-port" counter

v1 of our mkUpDownSatCounter may be functionally correct, but it's hardly a "2-port" counter!

When we say "2-port", we are making a performance characterization, i.e., we expect both ports to be operable in the same clock.

For this, we need to replace the Reg in mkUpDownSatCounter with an EHR, a different primitive that allows "multiple reads and writes" within a clock.

bluespec

First: specifying the semantics of the two ports

But before we worry about implementations and EHRs, we must first specify the *semantics* of the two ports! Specifically:

When both countA and countB are operated in the same clock,
- what should be the final value of the counter?
- what should be the "old" values returned by each method?

In light of the finite precision arithmetic, and the saturating behavior, there is no obvious unique answer! It is a design choice!

In RTL IP, this is typically where you'll see an *ad hoc* choice made by the designer
- Which is (hopefully!) implemented correctly
- Which is (hopefully!) documented clearly and fully in English text in the datasheet
- Which may contain usage rules that the user of the IP must follow

In BSV, method orderings give us a formal and precise way to specific the semantics. By specifying "countA < countB" or "countA > countB", we give precise answers to the above two semantic questions, because when operated in the same clock, there is a well-defined *logical* ordering that specifies the behavior.

Further, *bsc* always checks correct usage because it's in the semantics, not *ad hoc* English

bluespec

EHRs (Ephemeral History Registers)

An EHR offers a *vector* of standard Reg interfaces that can be operated concurrently:

```
interface EHR #(numeric type n, type t);
   interface Vector #(n, Reg #(t)) ports;
endinterface
```

BSV notes:
- "Vector" is a standard importable BSV library
- The parameter n is the number of elements in the vector; t is the data type EHR

The ports of an EHR can be operated concurrently, with the following ordering constraints:

```
ports [0]._read < ports [0]._write  <
ports[1]._read  <  ports[1]._write  <
ports[2]._read  <  ports[2]._write  <
…  …
ports[n-1]._read < ports[n-1]._write
```

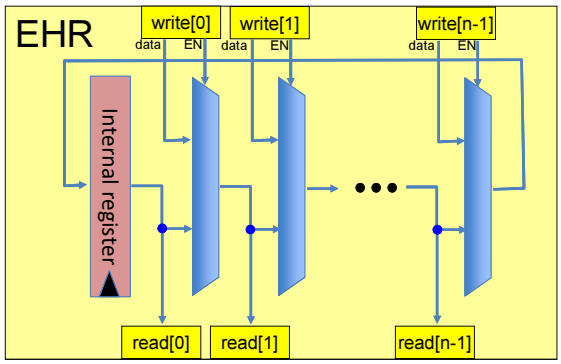
This is the same as the standard register method-ordering constraint

But note that a value written in port 0 can be read concurrently (by a logically later rule in the same clock) on port 1, unlike an ordinary register where a write can only be read in the next clock

bluespec

A possible implementation of an EHR

This figure shows a possible circuit implementation of an HER:



But it is important, once again, to keep separate the logical semantics from any implementation semantics.

When using EHRs in BSV, one only needs to consider its method-ordering constraints (shown in the previous slide).

bluespec

An implementation using EHRs (v2)

```
module mkUpDownSatCounter (UpDownSatCounter_Ifc);
   EHR #(2, Int #(4)) ctr <- mkEHR (0);     // 2 ports

   function ActionValue #(Int #(4)) fn_count (Integer p, Int #(4) delta);
      actionvalue
 // Extend the precision to avoid over/under flows
 Int #(5) new_val = extend (ctr.ports [p]) + extend (delta);
 if (new_val > 7) ctr.ports [p] <= 7;
 else if (new_val < -8) ctr.ports [p] <= -8;
 else ctr.ports [p] <= truncate (new_val);

 return ctr.ports [p];    // note: returns old value
      endactionvalue
   endfunction

   method countA (Int #(4) delta) = fn_count (0, delta);
   method countB (Int #(4) delta) = fn_count (1, delta);
endmodule
```

For "countA < countB".

To implement "countB < countA", change to:

```
… = fn_count (1, delta);
… = fn_count (0, delta);
```

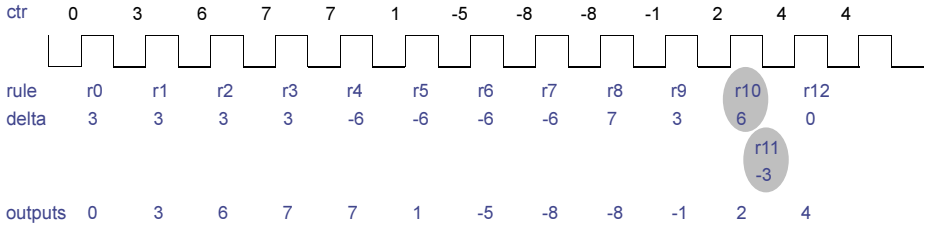
This is only a slight change to v1:

- The internal "ctr" is now a 2-port EHR instead of a Reg
- fn_count is now parameterized by the EHR port "p" it should use
- countA and countB call this function with ports 0 and 1, respectively, thereby implementing the ordering semantics "countA < countB"

bluespec

Behavior and outputs for v2

We expect the following behavior (r10 < r11 in same clock):



When we run the program (v2), we see:

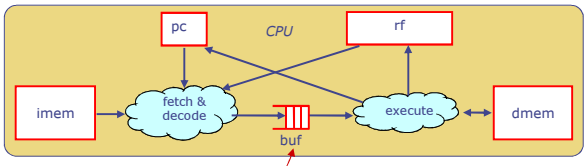
```
cycle 1, r0: is 0, count (3)
cycle 2, r1: is 3, count (3)
cycle 3, r2: is 6, count (3)
cycle 4, r3: is 7, count (3)
cycle 5, r4: is 7, count (-6)
cycle 6, r5: is 1, count (-6)
cycle 7, r6: is -5, count (-6)
cycle 8, r7: is -8, count (-6)
cycle 9, r8: is -8, count (7)
cycle 10, r9: is -1, count (3)
cycle 11, r10: is 2, count (6)
cycle 11, r11: is 7, count (-3)
cycle 12, r12: is 4, count (0)
```

same cycle

bluespec

A second example

Recall our 2-stage CPU pipeline from a previous lecture:



And let us focus on the FIFO connecting the stages:

Usually this is just a 1-element FIFO (we call it a "PipelineFIFO").

a.k.a. "pipeline register with interlock" (the interlock is just the extra valid bit that allows the execute stage to stall if there is nothing in the pipeline register, and allows the fetch/decode stage to stall if there is already something in the register which has not been consumed by the execute stage).

bluespec

An implementation using ordinary registers (v1)

```
module mkFIFOF1 (FIFOF #(t));
   Reg #(t)          rg       <- mkRegU;      // data storage
   Reg #(Bit #(1))   rg_count <- mkReg (0);  // # of items in FIFO (0 or 1)

   method Bool notEmpty = (rg_count == 1);
   method Bool notFull  = (rg_count == 0);

   method Action enq (t x) if (rg_count == 0);  // can enq if not full
      rg <= x;
      rg_count <= 1;
   endmethod

   method t first ()  if (rg_count == 1);  // can see first if not empty
      return rg;
   endmethod

   method Action deq () if (rg_count == 1);  // can deq if not empty
      rg_count <= 0;
   endmethod

   method Action clear;
      rg_count <= 0;
   endmethod
endmodule
```

But: enq and {first, deq} could never be concurrent, with mutually exclusive conditions: rg_count == 0 and rg_count == 1

Implication ➔ the fetch/decode stage and and the execute stage in the 2-stage CPU pipeline could never execute in the same clock (it isn't really a pipeline!)

bluespec

First: specifying the semantics concurrent FIFOs

But before we worry about implementations, we must first specify the *semantics* of concurrency on FIFO methods. In BSV we commonly use the following two kinds of FIFOs:

PipelineFIFOs:
- When empty, only enq is enabled
- When full, enq, first and deq are enabled, with: {first,deq} < enq
  i.e., if both methods are enabled, logically it is like {first,deq} followed by enq,
  i.e., data currently in the FIFO is returned for {first,deq}, and new data is enqueued.

BypassFIFOs:
- When full, only {first,deq} is enabled
- When empty, enq, first and deq are enabled, with: enq < {first,deq}
  i.e., if both methods are enabled, logically it is like enq followed by {first,deq},
  i.e., the newly enqueued value is "bypassed" through to {first,deq}.

bluespec

An implementation of Pipeline FIFOs using EHRs

```
module mkPipelineFIFOF (FIFOF #(t));
   EHR #(3, t)                ehr       <- mkEHRU;     // data storage
   EHR #(3, Reg #(Bit #(1)))  ehr_count <- mkEHR (0);  // # of items in FIFO

   method Bool notEmpty = (ehr_count.ports[0] == 1);
   method Bool notFull  = (ehr_count.ports[1] == 0);

   method Action enq (t x) if (ehr_count.ports[1] == 0);
      ehr.ports[1] <= x;
      ehr_count.ports[1] <= 1;
   endmethod

   method t first ()  if (ehr_count.ports[0] == 1);
      return ehr.ports[0];
   endmethod

   method Action deq () if (ehr_count.ports[0] == 1);
      ehr_count.ports[0] <= 0;
   endmethod

   method Action clear;
      ehr_count.ports[2] <= 0;
   endmethod
endmodule
```

This is only a slight change to v1:

- notEmpty, first and deq use EHR port 0
- notFull and enq use EHR port 1
- clear uses EHR port 2

bluespec

An implementation of BypassFIFOs using EHRs

```
module mkBypassFIFOF (FIFOF #(t));
   EHR #(3, t)                ehr       <- mkEHRU;     // data storage
   EHR #(3, Reg #(Bit #(1)))  ehr_count <- mkEHR (0);  // # of items in FIFO

   method Bool notEmpty = (ehr_count.ports[1] == 1);
   method Bool notFull  = (ehr_count.ports[0] == 0);

   method Action enq (t x) if (ehr_count.ports[0] == 0);
      ehr.ports[0] <= x;
      ehr_count.ports[0] <= 1;
   endmethod

   method t first ()  if (ehr_count.ports[1] == 1);
      return ehr.ports[1];
   endmethod

   method Action deq () if (ehr_count.ports[1] == 1);
      ehr_count.ports[1] <= 0;
   endmethod

   method Action clear;
      ehr_count.ports[2] <= 0;
   endmethod
endmodule
```

This is only a slight change to v1:

- notFull and enq use EHR port 0
- notEmpty, first and deq use EHR port 1
- clear uses EHR port 2

bluespec

EHR summary

The EHR is a highly concurrent primitive, i.e., it has multiple methods that can be invoked by multiple rules within a clock in a well-defined logical sequential order.

When using an EHR to communicate within rules that you want to be concurrent (i.e., able to fire in the same clock),

- first, be clear about what semantics you want, by thinking about what logical ordering of rules you want
- then, use EHRs to implement that ordering
  - (ascending EHR port indexes directly correspond to ordering)

In the pure rule semantics, where we don't worry about mapping to clocks, EHRs are just like ordinary registers: port indexes are irrelevant, and we could just replace each EHR by an ordinary Reg.

In practice, it turns out that we don't use EHRs explicitly very much. Most often, higher-level library IP (like PipelineFIFO or BypassFIFO) is what we want. Or, we may create useful IPs like this (such as a 2-port counter) using EHRs, for subsequent use.

bluespec

The rest of the slides in this lecture are optional

They describe a lower-level primitive called RWires
(and a few of its variations)

(in fact, EHRs are not BSV primitives, but implemented within BSV itself using RWires)

RWires (and their variants) are often used "at the edge of" a BSV design, where it interfaces to existing Verilog IP

bluespec

RWires

The most general form of "wire" family is the RWire interface and mkRWire primitive module:

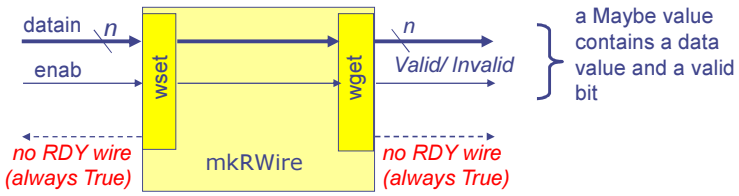
```
interface RWire #(type  t);
   method Action     wset (t datain);
   method Maybe#(t)  wget;
endinterface

module mkRWire (RWire#(t));   // primitive
```

Ordering constraint:
wset < wget

Suppose rule rA invokes rw.wset (x)
Then, in rule rB (logically later in the schedule):

- if (rw.wget matches tagged Valid .x) then rB knows that rA is firing in this clock and communicating the value x
- if (rw.wget matches tagged Invalid) then rB knows that rA is not firing in this clock

Implementation:



bluespec

PulseWires

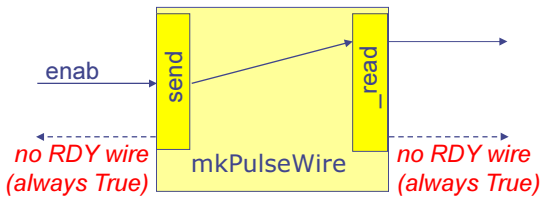
The PulseWire interface and mkPulseWire module is a special case of RWires where there is no value to be communicated

```
interface PulseWire;
    method Action send;
    method Bool   _read;
endinterface

module mkPulseWire (PulseWire);   // primitive
```

Ordering constraint:
send < _read

The _read method returns True if the send method is being invoked, else returns False

Implementation:



bluespec

The Wire interface and mkDWire module

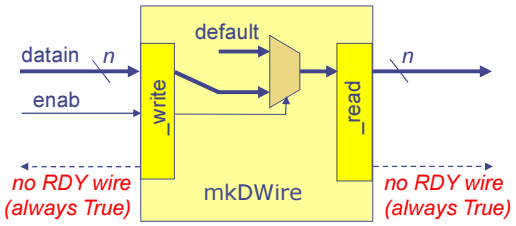
The Wire interface is just a synonym for the Reg interface.

The mkDWire module is a "primitive" (D for default)

```
typedef  Reg#(t)  Wire#(type t);



module mkDWire #(t default) (Wire#(t));
  ... primitive imported Verilog ...
endmodule
```

Ordering constraint:
_write < _read
Note! opposite of registers!

Implementation:



bluespec

bluespec

End

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