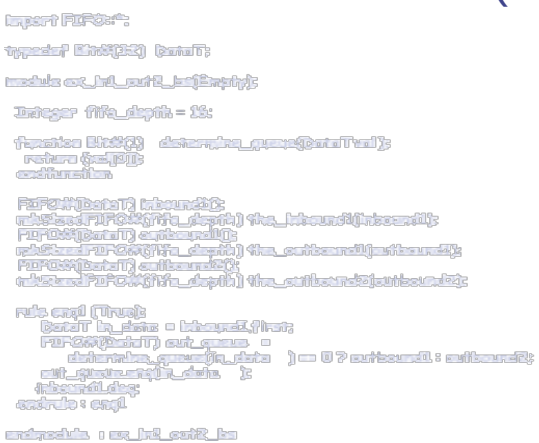
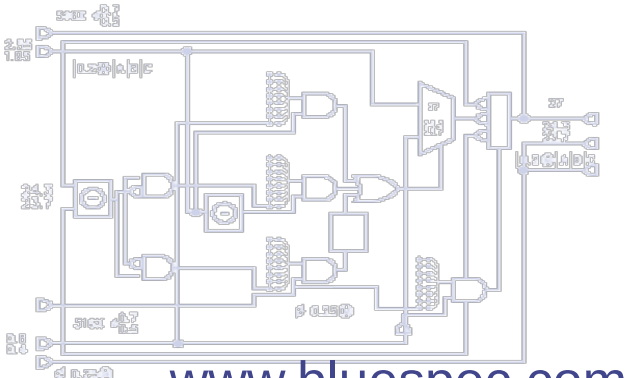
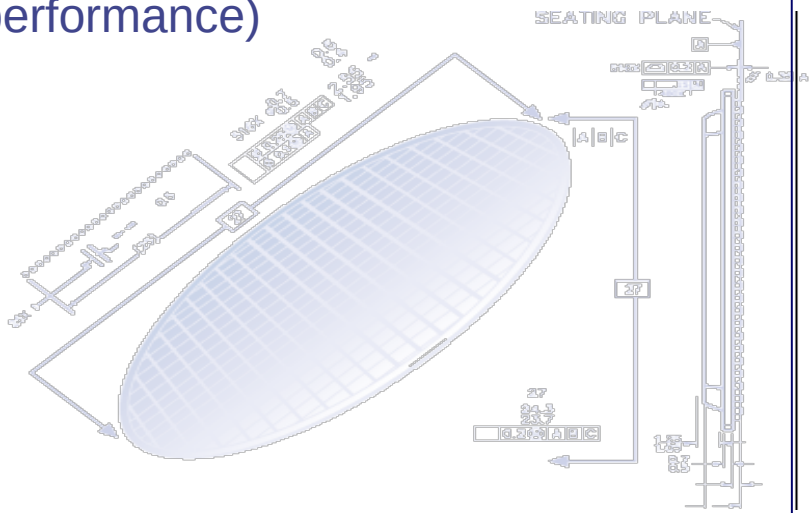
Bluespec Training

Lec_CRegs

A CReg (Concurrent Register) is a register-like primitive that enables greater concurrency (more rules per clock, so greater performance)



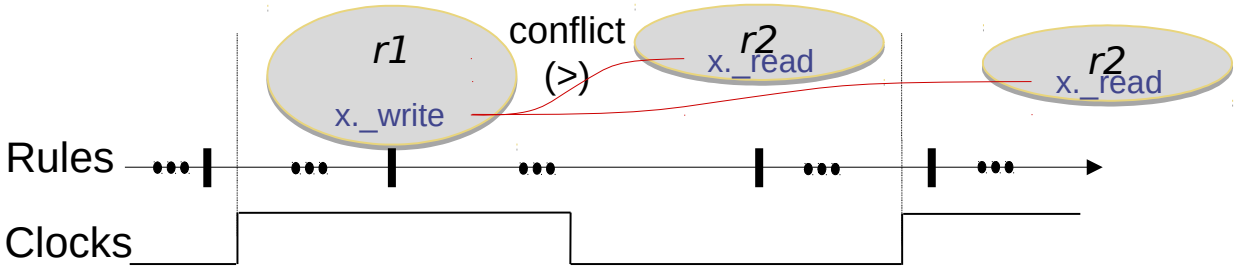




www.bluespec.com

Enabling greater concurrency

With a primitive module like mkReg, the effect of a rule's Action (_write) is not visible until the next clock (via _read), because of its method ordering constraint (<)



For greater rule concurrency, we need another primitive whose method ordering constraints allow an Action's effect to be visible in the same clock.

In Bluespec, we use a primitive called the *CReg*[1].

[1] "CReg" = Concurrent Register. These are based on the "Ephemeral History Register" which was researched by Daniel Rosenband at MIT in 2004.

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 =
   countA :: Int 4 -> ActionValue (Int 4)
   countB :: Int 4 -> ActionValue (Int 4)
```

- The “two ports” are the two identical 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 like in conventional arithmetic; so, the order of these operations matters here!

An implementation using ordinary registers (v1)

```
mkUpDownSatCounter :: Module UpDownSatCounter_Ifc
mkUpDownSatCounter =
  module
    ctr :: Reg (Int 4) ← mkReg 0

    let fn_count :: Int 4 -> ActionValue (Int 4)
        fn_count    delta = do
            // Extend the precision to avoid over/under flows
            new_val :: Int 5 = extend ctr + extend delta
            ctr := if (new_val > 7) then 7
                   else if (new_val < -8) then -8
                   else truncate new_val
            return ctr    // note: returns old value

    interface
        countA  deltaA = fn_count  delta_A
        countB  deltaB = fn_count  delta_B
```

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

Bluespec notes:

- “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

A testbench to drive the up/down counter module

```
mkTest :: Module Empty
mkTest =
  module
    ctr   :: UpDownSatCounter_Ifc <- mkUpDownSatCounter
    step  :: Reg int  <- mkReg 0
    flag0 :: Reg Bool <- mkReg False; flag1 :: Reg Bool <- mkReg False

    count_show :: Integer -> Bool ->  Int 4 -> Action
    count_show    rulenum    a_not_b  delta = do
        x <- if a_not_b then ctr.countA  delta else ctr.countB delta
        $display "cycle %0d, r%0d: is %0d, count (%0d)”  cur_cycle  rulenum  x  delta

    -- Rules 0-9 are sequential, just testing one method at a time
    "r0": when (step == 0) ==> do { count_show  0  True  3 ; step := 1 }
    "r1": when (step == 1) ==> do { count_show  1  True  3 ; step := 2 }
                  … and similarly, sequentially feed deltas of  3,3, -6,-6,-6,-6, 7, 3,
    -- Concurrent execution
    "r10": when (step == 10 && not flag0) ==> do { count_show 10 True  6;       flag0 := True }
    "r11": when (step == 10 && not flag1) ==> do { count_show 11 False (neg 3); flag1 := True }

    -- Show final value
    "r12": when (step == 10 && flag0 && flag1) ==> do { count_show 12 True 0; $finish }
```

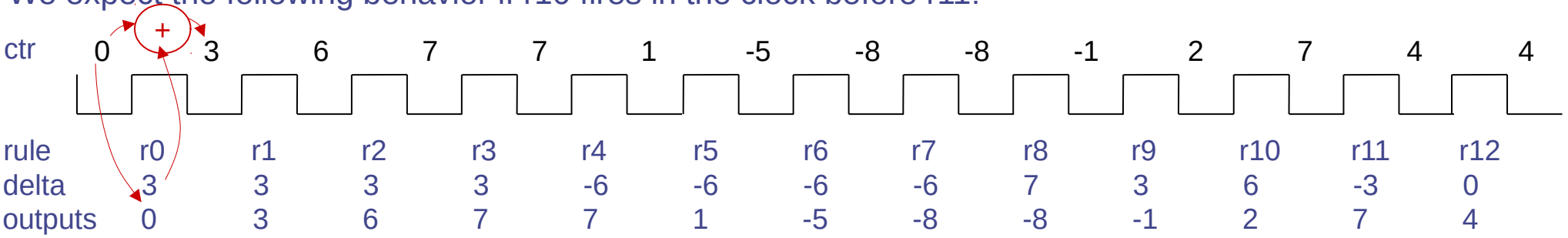
In rules 0-9, we call either countA or countB with deltas: 3,3,3,3, -6,-6,-6,-6, 7, 3
The rule conditions and step assignments force them to fire 1 rule per clock (and so it doesn't matter whether we call countA or countB in these rules).

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

Rule 12 just displays the final counter value and exits.

Expected behavior and outputs for v1

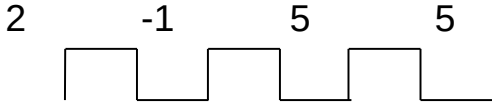
We expect the following behavior if r10 fires in the clock before r11:



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

Actual output for v1

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

```
Warning: "Test.bs", 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
  (countA and countB conflict because they both read and write the “ctr” register inside mkUpDownSatCounter, thus both possible rule orders will violate a “_read < _write” ordering constraint)
- *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)
- Note: you can force the opposite priority by joining the rules with
  rJoinDescendingUrgency

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)
```

2 cycles

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 CReg, a different primitive that allows “multiple reads and writes” within a clock.

First: specifying the semantics of the two ports

Before we worry about implementations and CRegs, we must first specify the *desired 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 designs, 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 the user of the IP must follow, and which therefore need verification

In Bluespec, method orderings give us a formal and precise way to specific the semantics. By specifying that we want "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 exactly.

CRegs (Concurrent Registers)

A CReg provides an *array* of standard Reg interfaces that can be operated concurrently:

```
interface CReg t = Array (Reg t)
```

"Array" is a standard type in the Bluespec library
Unfortunately array-selection is not so convenient in Bluespec Classic, so we provide some convenience functions to access CReg array components. These are defined in Resources/CReg_Classsic.bsv

```
select_CReg :: CReg t -> Integer -> Reg t

read_CReg   :: CReg t -> Integer -> t
```

The ports (individual Reg interfaces in the array) of an CReg can be operated concurrently, with the following ordering constraints:

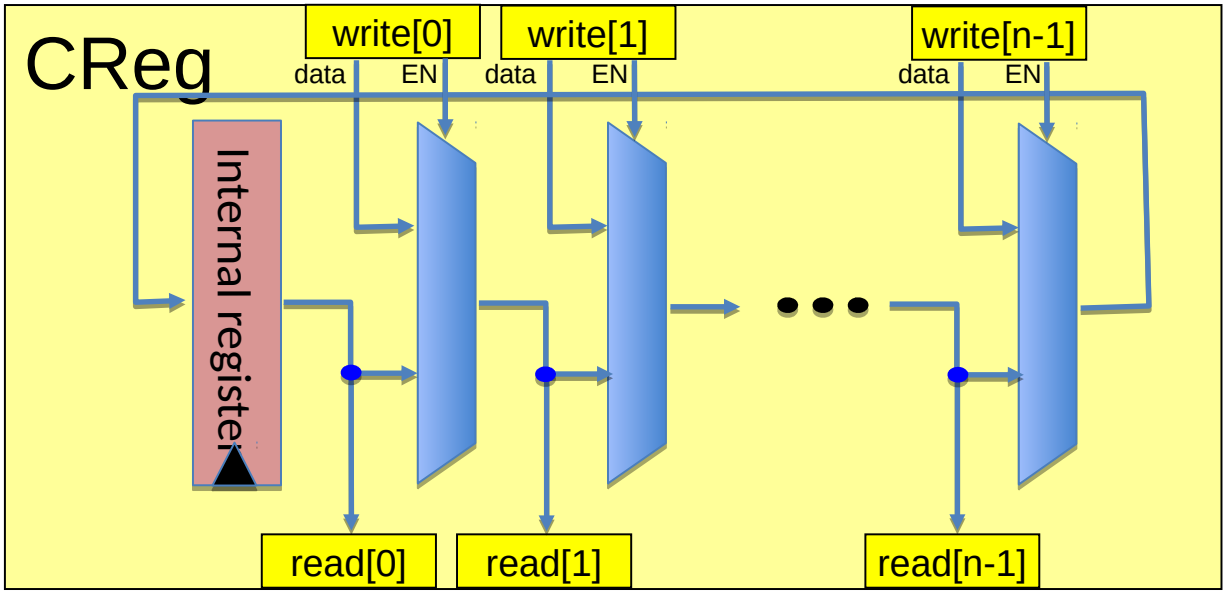
ports [0]._read <= ports [0]._write <
ports[1]._read <= ports[1]._write <
ports[2]._read <= ports[2]._write <
… …
ports[n-1]._read <= ports[n-1]._write

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

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

A possible implementation of an CReg

This figure shows a possible circuit implementation of a CReg:



But note, this is not a *definition* of an CReg, it is merely shown to strengthen intuition. It is important, as usual, to keep separate the logical semantics from any implementation semantics. When using CRegs in Bluespec, one only needs to consider its method-ordering constraints (shown in the previous slide).

Implementing our counter using CRegs (v2)

```
mkUpDownSatCounter :: Module UpDownSatCounter_Ifc
mkUpDownSatCounter =
  module
    ctr :: Array  (Reg  (Int 4)) <- mkCReg 2 0    // 2 ports, intial 0

    fn_count :: Integer -> Int 4 -> ActionValue (Int 4)
    fn_count    p          delta = do
        -- Extend the precision to avoid over/under flows
        new_val :: Int 5  = extend (read_CReg  ctr.ports  p) + extend delta
        select_CReg ctr p := if (new_val > 7) then 7
                             else if (new_val < -8) then -8
                             else truncate  new_val

        return  read_CReg  ctr  p    // note: returns old value

    interface
        countA  delta = fn_count  0  delta
        countB  delta = fn_count  1  delta
```

Change Reg to CReg

Add CReg port selections to reads and writes

For "countA < countB".

To implement "countB < countA", change to:

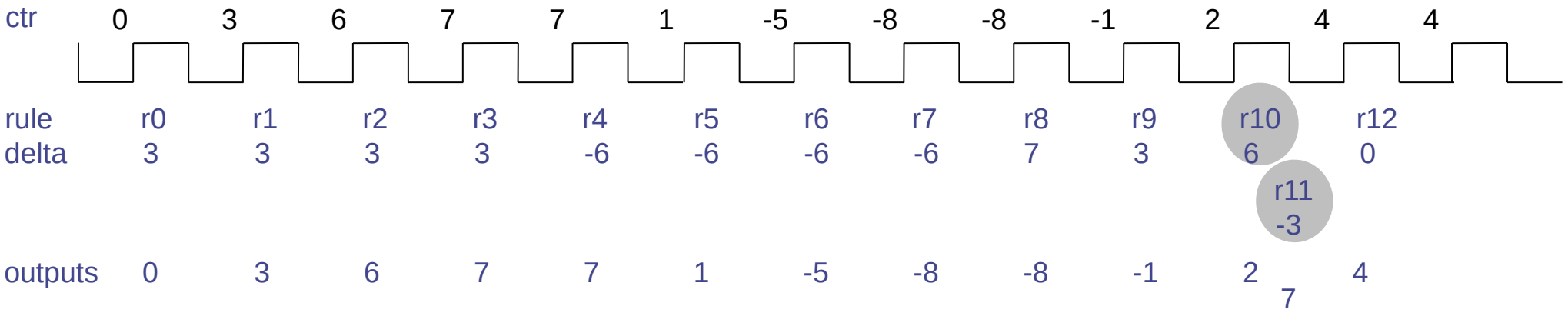
```
… = fn_count  1  delta
… = fn_count  0  delta
```

This is only a slight change to v1:

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

Behavior and outputs for v2

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



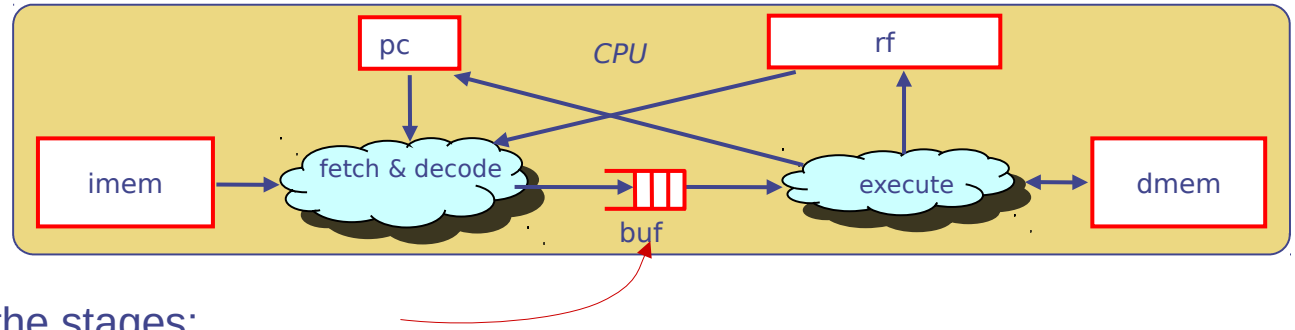
When we run the program (v2), we see:

```
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

A second example

Consider a 2-stage CPU pipeline:



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).

An implementation using ordinary registers (v1)

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

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

    enq x = do
              rg := x
              rg_count := 1
      when (rg_count == 0)    -- can enq if not full

    first = rg
      when (rg_count == 1)    -- can see first if not empty

    deq = rg_count := 0
      when (rg_count == 1)    -- can deq if not empty

    clear = rg_count := 0
```

```
interface FIFOF t =
    notEmpty :: Bool
    notFull  :: Bool
    enq      :: t -> Action
    first    :: t
    deq      :: Action
    clear    :: Action
```

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!)

First: specify desired semantics of concurrent methods

Before we worry about implementations, we must first specify the desired *semantics* of concurrency on FIFO methods. In Bluespec 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}.

An implementation of Pipeline FIFOs using CRegs

```
mkPipelineFIFOF :: Module (FIFOF t)
mkPipelineFIFOF =
  module
    crg ::       Array  (Reg  t)       <- mkCRegU 3
    crg_count :: Array  (Reg  (Bit 1)) <- mkCReg  3  0  // # of items in FIFO; init 0

    notEmpty = (read_CReg  crg_count  0 == 1)
    notFull  = (read_CReg  crg_count  1 == 0)

    enq  x = do
       (select_CReg  crg  1)       := x
       (select_CReg  crg_count  1) := 1
      when (read_CReg  crg_count 1 == 0)

    first = read_CReg  crg  0
      when (read_CReg  crg_count 0 == 1)

    deq = select_CReg  crg_count 0 := 0
      when (read_CReg  crg_count 0 == 1)

    clear = do
        (select_CReg  crg_count 2) := 0
```

This is only a slight change to v1:

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

An implementation of BypassFIFOs using CRegs

```
mkBypassFIFOF :: Module (FIFOF t)
mkBypassFIFOF =
  module
    crg ::       Array  (Reg  t)       <- mkCRegU 3
    crg_count :: Array  (Reg  (Bit 1)) <- mkCReg  3  0  // # of items in FIFO; init 0

    notEmpty = (read_CReg  crg_count  1 == 1)
    notFull  = (read_CReg  crg_count  0 == 0)

    enq  x = do
       (select_CReg  crg  0)       := x
       (select_CReg  crg_count  0) := 1
      when (read_CReg  crg_count 0 == 0)

    first = read_CReg  crg  1
      when (read_CReg  crg_count 1 == 1)

    deq = select_CReg  crg_count 1 := 0
      when (read_CReg  crg_count 1 == 1)

    clear = do
        (select_CReg  crg_count 2) := 0
```

This is only a slight change to v1:

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

CReg summary

The CReg 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 a CReg to communicate between 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 CRegs to implement that ordering
  - (ascending CReg port indexes directly correspond to ordering)

Note: a design using CRegs will be functionally correct with any schedule, even one rule per clock. In the extreme schedule of one rule per clock, an CReg is exactly equivalent to an ordinary register (using mkReg).

In practice, we more often directly use concurrent library modules like PipelineFIFO and BypassFIFO. If necessary, we use CRegs to implement a concurrent module that is not available in the library.

Hands-on

- BSV-by-Example book: Examples in Chapter 8

bluespec

End

Questions?

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