

# Learn RISC-V CPU Implementation and BSV

(BSV: a High-Level Hardware Design Language)

Rishiyur S. Nikhil

L17: **BSV**: Tighter Rule scheduling with CRegs



# Reminders

Please git clone: [https://github.com/rsnikhil/Learn\\_Bluespec\\_and\\_RISCV\\_Design](https://github.com/rsnikhil/Learn_Bluespec_and_RISCV_Design)  
(git pull for latest version). Repository structure:

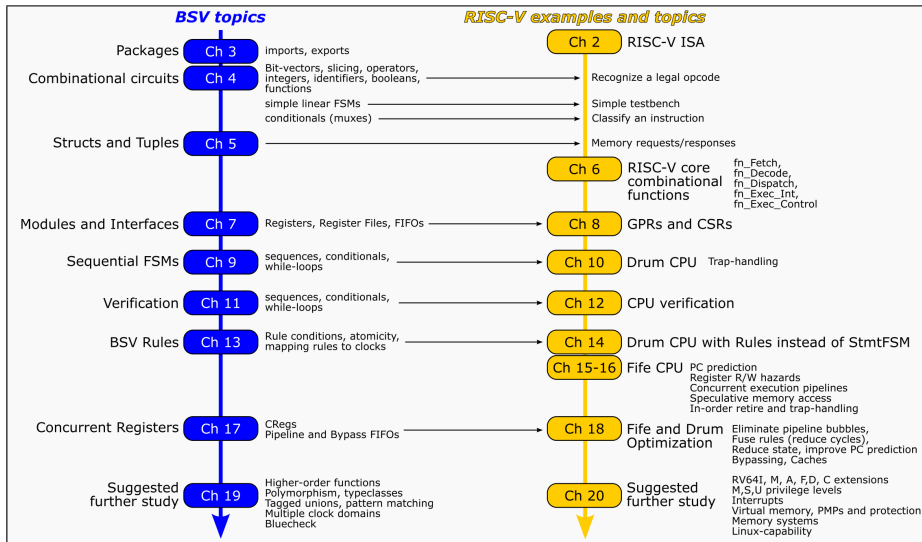
```
./Book_BLang_RISCV.pdf
  Slides/
    Slides_01_Intro.pdf
    Slides_02_ISA.pdf
    ...
  Exercises/
    Ex-03-A-Hello-World/
    Ex-03-B-Top-and-DUT/
    ...
  Code/
    src_Top/
    src_Drum/
    src_Fife/
    src_Common/
    ...
  Doc/Installing_bsc_Verilator_etc.{adoc,html}
```

- Slides and Exercise are numbered in sync with book Chapter numbers.
- For Exercises, please see Appendix E of the book. Some (not all) exercises have associated code in the Exercises/ directory.

To compile and run the code for exercises, Drum and Fife, please make sure you have installed:

- bsc compiler (see <https://github.com/B-Lang-org/bsc>)
- Verilator compiler (see <https://www.verilator.org/>)

# Chapter Roadmap



# Table of Contents

## 1 Reminders

## 2 Motivation

- Motivational Example

## 3 Concurrent Registers

- CRegs
- Hardware intuition
- CReg ordering properties
- Up-down counter with CReg
- Example: CSR `mcycle` in RISC-V

## 4 SpecialFIFOs

- PipelineFIFOs
- BypassFIFOs
- Back-to-back BypassFIFO+PipelineFIFO

## 5 Final Comments

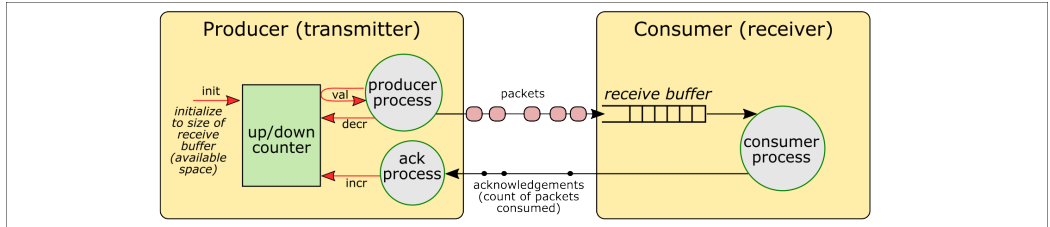
# BSV: Tighter Rule Scheduling: Motivation

# Tighter Rule Scheduling: Motivation

	BSV	Analogy: RISC-V ISA
(A)	Rule semantics are abstract: one rule at a time	<i>RISC-V ISA semantics are abstract: one instruction at a time</i>
(B)	<p>When the <i>bsc</i> compiler maps a set of rules to execute simultaneously (in the same clock), all their methods occur at the same instant (clock edge). All “read-values” are from the previous clock edge; all “write-values” (Action and ActionValue results) are visible only at the next clock edge.</p> <p><i>bsc</i> must ensure that the <i>ordering</i> of methods in (B) is consistent with (A).</p> <p>In particular, <i>bsc</i> may introduce (combinational) control circuits to <i>stall</i> a rule to avoid violating (A).</p>	<p><i>RISC-V instructions can be executed in pipelines, in parallel (superscalar), out-of-order, ... The implementation must ensure that the order in which they retire, and the order in which they read and write registers and memory, is consistent with (A).</i></p> <p><i>In particular, an instruction may be stalled to avoid violating (A).</i></p>

# Tighter Rule Scheduling: Motivational Example: Up-Down Counter (1/2)

Consider this scenario, where a Producer streams packets over a long connection to a Consumer:



To avoid over-running the receive buffer, The producer:

- initializes the counter to the available space in the buffer;
- for every packet sent, decrements the counter (**decr** method);
- stalls (doesn't send) if the counter value (**val** method) is zero (no space available).

As the receiver consumes packets from the buffer, it sends acknowledgements back to indicate the amount of space freed.

In the transmitter, acknowledgements are incremented back into the counter (**incr** method), allowing transmission to continue.

```
interface Up_Down_Counter_IFC;
    method Action    init (Bit #(4) init_val);
    method Bit #(4)  val;
    method Action    decr;
    method Action    incr;
endinterface
```

# Tighter Rule Scheduling: Motivational Example: Up-Down Counter (2/2)

Here is a possible implementation of the “Up-Down” Counter:

```
module mkUp_Down_Counter_I (Up_Down_Counter_IFC);
  // STATE
  Reg #(Bit #(4)) rg_counter <- mkReg (15);

  // -----
  // INTERFACE

  method Action init (Bit #(4) init_val);
    rg_counter <= init_val;
  endmethod

  method Bit #(4) val;
    return rg_counter;
  endmethod

  method Action decr () if (rg_counter != 0);
    rg_counter <= rg_counter - 1;
  endmethod

  method Action incr () if (rg_counter != 15);
    rg_counter <= rg_counter + 1;
  endmethod
endmodule
```

Analysis: Rules invoking `.incr()` and `.decr()` cannot execute on the same clock edge.

- (A) In the abstract rule semantics, either `.incr()` precedes `.decr()` or *vice versa*.

In either case, the latter rule observes the update from the previous rule.

- (B) If they executed on the same clock, neither rule sees the update from the other rule. This is inconsistent with (A).

Thus, this implementation cannot send a packet (`decr`) and register an ack (`incr`) in the same clock, even though the two streams are asynchronous and concurrent.

On the average, we can only send (and register ack) on every alternate clock.

**Solution:**

Use a *Concurrent Register* (CReg) for `rg_counter`.



# BSV: Concurrent Registers (CRegs)

# CRegs

A Concurrent Register, or CReg, is a module provided by the *bsc* library. Its interface is an *array* of *Reg#(t)* interfaces:

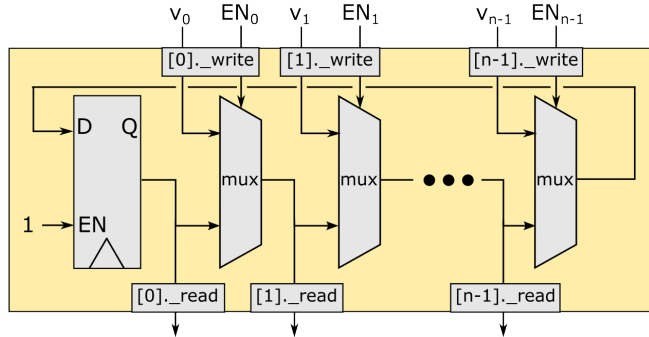
```
module mkCReg #(parameter Integer n,  
               parameter a_type resetval)  
  (Reg#(a_type) ifc[])
```

It is instantiated similar to this example:

(This example: register content type: *Bit#(4)*; initial value: 15; two *Reg* interfaces).

```
// parameter n is 2, resetval is 15  
Array #(Reg #(Bit #(4))) crg_counter <- mkCReg (2, 15);
```

# Hardware intuition for a CReg



This has an array of  $n$  Reg interfaces, indexed from 0 to  $n-1$ .  
The  $j$ 'th interface has  $[j]_{\_read}()$  and  $[j]_{\_write}()$  methods.

# CReg ordering properties

- All the methods can be invoked in the same clock.
- A read at the  $j$ 'th register interface, i.e.,  $x[j].\_read$  returns the latest of:
  - the value  $v_{j-1}$ , if  $x[j-1].\_write(v_{j-1})$  is being invoked;
  - else the value  $v_{j-2}$ , if  $x[j-2].\_write(v_{j-2})$  is being invoked;
  - ...
  - else the value  $v_1$ , if  $x[1].\_write(v_1)$  is being invoked;
  - else the value  $v_0$ , if  $x[0].\_write(v_0)$  is being invoked;
  - else the value in the register.
- The register value is updated with the latest of:
  - the value  $v_{n-1}$ , if  $x[n-1].\_write(v_{n-1})$  is being invoked;
  - else the value  $v_{n-2}$ , if  $x[n-2].\_write(v_{n-2})$  is being invoked;
  - ...
  - else the value  $v_1$ , if  $x[1].\_write(v_1)$  is being invoked;
  - else the value  $v_0$ , if  $x[0].\_write(v_0)$  is being invoked;
  - else the current value in the register.

This ordering corresponds exactly to the left-to-right and top-to-bottom ordering of the methods in the hardware-intuition diagram (Slide 11).

# Up-down counter with a CReg

```
module mkUp_Down_Counter_I (Up_Down_Counter_IFC);
  // STATE
  Array #(Reg #(Bit #(4))) crg_counter <- mkCReg (3,15);

  // -----
  // INTERFACE
  method Action init (Bit #(4) init_val);
    crg_counter [2] <= init_val
  endmethod

  method Bit #(4) val;
    return rg_counter [0];
  endmethod

  method Action decr if (rg_counter != 0);
    rg_counter [1] <= rg_counter [1] - 1;
  endmethod

  method Action incr if (rg_counter != 15);
    rg_counter [0] <= rg_counter [0] + 1;
  endmethod
endmodule
```

Some questions to ponder:

- What does method `val` return?
  - The original value in the register?
  - The value after the increment?
  - The value after the decrement?
  - The value after the increment and decrement?
- What happens if methods `incr` and `decr` are called simultaneously? Which one happens (semantically) “first”?
  - Note: `incr` saturates at 15, and `decr` saturates at 0, so the order matters!

*Hint:* The answers are in the choice of CReg indexes in each method.

# Example: CSR `mcycle` in RISC-V

CSR `mcycle` (RISC-V CPU cycle counter) is updated by two “processes”:

- (A) Standalone infinite loop incrementing `mcycle` on every cycle.
- (B) Instruction execution, when we have a `CSRxx` instruction that writes to `mcycle`.

The RISC-V spec says that when (B) happens, it should override (A).

We can instantiate a CReg for this:

```
_____ from src_Common/CSRs.bsv _____  
Array #(Reg #(Bit #(64))) csr_mcycle <- mkCReg (2, 0);
```

(A) Standalone process incrementing it:

```
_____ from src_Common/CSRs.bsv _____  
rule rl_count_cycles;  
  csr_mcycle [0] <= csr_mcycle [0] + 1;  
endrule
```

(B) `CSRxx` instruction execution (overrides (A) because of higher CReg index):

```
_____ from src_Common/CSRs.bsv in function fav_csr_write() _____  
  csr_mcycle [1] <= csr_val;
```

# BSV: Higher-performance FIFOs (in library SpecialFIFOs)

(implemented using Concurrent Registers)

# BSV: a FIFO implemented with ordinary registers

Consider the following module implementing a 1-element FIFO with a FIFOF interface, using ordinary registers (mkReg, mkRegU):

```
module mkFIFO (FIFO #(Bit #(32)));
  Reg #(Bit #(32)) rg_data <- mkRegU;
  Reg #(Bool)      rg_full <- mkReg (False);

  // -----
  // INTERFACE

  method Bool notEmpty ();
    return rg_full;
  endmethod

  method Bit #(32) first () if (rg_full);
    return rg_data;
  endmethod

  method Action deq () if (rg_full);
    rg_full <= False;
  endmethod
  ...
endmodule
```

```
...
method Bool notFull ();
  return (! rg_full);
endmethod

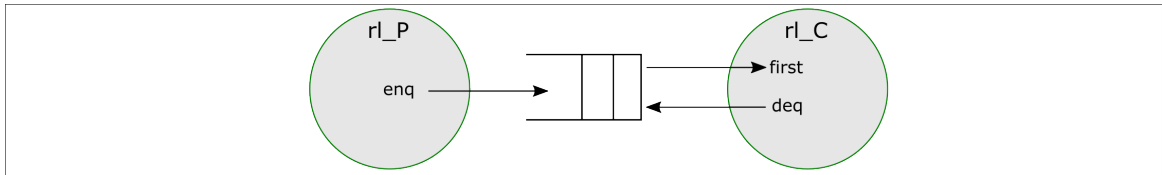
method Action enq (Bit #(32) x) if (! rg_full);
  rg_data <= x;
  rg_full <= True;
endmethod

method Action clear;
  rg_full <= False;
endmethod
endmodule
```



# BSV: Analysis of the performance of mkFIFOF

Consider a producer rule `r1_P` that invokes `enq`, and a consumer rule `r1_C` that invokes `first` and `deq`:



These rules cannot fire at the same instant (on the same clock) because both of them read and write register `fg_Full` (because both `enq` and `deq` read and write the register).

We can use a CReg to relax this constraint, in two different ways, which we call `mkPipelineFIFOF` and `mkBypassFIFOF`, respectively.

# BSV: mkPipelineFIFO

Instead of mkReg for rg\_full, we can use mkCReg:

```
module mkFIFO (FIFO #(Bit #(32)));
module mkPipelineFIFO (FIFO #(Bit #(32)))
  Reg #(Bit #(t))      rg_data  <- mkRegU;
  Array #(Reg #(Bool)) crg_full <- mkCReg (3, False);

  // -----
  // INTERFACE

  method Bool notEmpty ();
    return crg_full [0];
  endmethod

  method Bit #(32) first () if (crg_full [0]);
    return rg_data;
  endmethod

  method Action deq () if (crg_full [0]);
    crg_full [0] <= False;
  endmethod
  ...
```

```
...
method Bool notFull ();
  return (! crg_full [1]);
endmethod

method Action enq (Bit #(32) x) if (! crg_full [1]);
  rg_data      <= x;
  crg_full [1] <= True;
endmethod

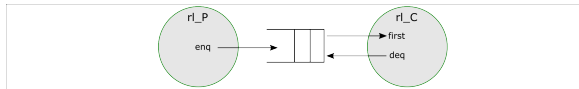
method Action clear;
  crg_full [2] <= False;
endmethod
endmodule
```

Note that the “dequeue” side methods use index [0], “enqueue” side use methods index [1]; and “clear” uses index [2].

These choices affect the ordering semantics of the methods.

# BSV: Analysis of the performance of mkPipelineFIFOF

Consider again a producer rule `r1_P` that invokes `enq`, and a consumer rule `r1_C` that invokes `first` and `deq`:



Because of `mkPipelineFIFOF`'s CReg indexes, in the equivalent rule-at-a-time semantics, `r1_C` fires *before* `r1_P`. Thus, even if the FIFO was full at the start of the clock, `r1_P` can still `enq` into the FIFO, *provided* `r1_C` is firing on the same clock.

Per the rule-at-a-time semantics, `r1_C` fires first, which empties the FIFO, *i.e.*, `r1_P` sees the FIFO as empty and is therefore able to `enq` into it.

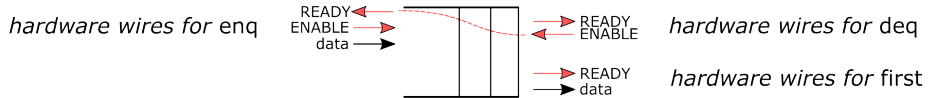
This is why we call it a “PipelineFIFO”: it is an ideal candidate for the FIFO between stages of a pipeline, allowing the downstream stage (the consumer) and the upstream stage (the producer) to fire on the same clock, advancing data in a pipelined manner.

Note also that another rule invoking `clear`, too, can fire on the same clock as `r1_P` and `r1_C`. Because of its CReg index, logically it fires “last”, and so will leave the FIFO in a finally empty state.

# BSV: Analysis of the hardware for mkPipelineFIFOF

In the Verilog for `mkPipelineFIFOF`, the `READY` signal for `enq` incorporates the `ENABLE` signal of the `deq` method (because if the FIFO is full from the previous clock, in this clock `enq` can only be invoked if `deq` is also being invoked).

Thus, there is a *combinational path* backward through the FIFO (a path that involves only wires and gates and no state-element) from the `deq` method to the `enq` method.



# BSV: mkBypassFIFO

What happens if we exchange the [0] and [1] CReg indexes?

```
Array #(Reg #(Bit #(32))) crg_data <- mkCRegU (2);
Array #(Reg #(Bool))      crg_full  <- mkCReg (3, False);

// -----
// INTERFACE

method Bool notEmpty ();
  return crg_full [1];
endmethod

method Bit #(32) first () if (crg_full [1]);
  return crg_data [1];
endmethod

method Action deq () if (crg_full [1]);
  crg_full [1] <= False;
endmethod
...
```

```
...
method Bool notFull ();
  return (! crg_full [0]);
endmethod

method Action enq (Bit #(32) x) if (! crg_full [0]);
  crg_data [0] <= x;
  crg_full [0] <= True;
endmethod

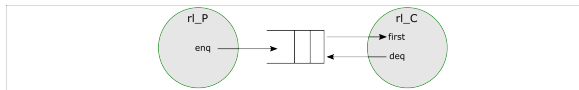
method Action clear;
  crg_full [0] <= False;
endmethod
endmodule
```

Now, the “enqueue” side methods use index [0] and the “dequeue” side methods use index [1] (“clear” still uses index [2]).

These choices change the ordering semantics of the methods.

# BSV: Analysis of the performance of mkBypassFIFO

Consider again a producer rule `r1_P` that invokes `enq`, and a consumer rule `r1_C` that invokes `first` and `deq`:



Because of `mkBypassFIFO`'s CReg indexes, in the equivalent rule-at-a-time semantics, `r1_C` fires *after* `r1_P`. Thus, even if the FIFO was empty at the start of the clock, `r1_C` can still `deq` from the FIFO, *provided* `r1_P` is firing on the same clock.

Per the rule-at-a-time semantics, `r1_P` fires first, which fills the empty FIFO, *i.e.*, `r1_C` sees the FIFO as full and is therefore able to `deq` from it.

This is why we call it a “BypassFIFO”. When used in a pipeline, the enqueued value can be “bypassed” straight through the FIFO to the consumer on the same clock.

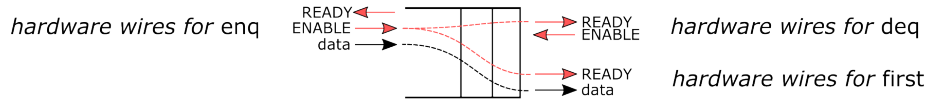
As with `mkPipelineFIFO`, another rule invoking `clear`, too, can fire on the same clock as `r1_P` and `r1_C`. Because of its CReg index, logically it fires “last”, and so will leave the FIFO in a finally empty state.

# BSV: Analysis of the hardware for mkBypassFIFOF

In the Verilog for `mkBypassFIFOF`, the `READY` signal for `first` and `deq` incorporates the `ENABLE` signal of the `enq` method (because if the FIFO is empty from the previous clock, in this clock `first` and `deq` can only be invoked if `enq` is also being invoked).

Similarly, for the data to be bypassed through, there has to be a combinational path from the `enq` argument to the `first` result.

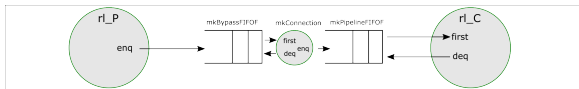
Thus, there are *combinational paths* forward through the FIFO (paths that involve only wires and gates and no state-element) from the `enq` method to the `first` and `deq` methods.



# BSV: Back-to-back composition of BypassFIFO and PipelineFIFO (1/2)

An interesting component is a back-to-back composition of the two high-performance FIFOs we have discussed.

Consider this code fragment, illustrated below:



```
module mk... (...);
  FIFOF #(Bit #(32)) f_bypass  <- mkBypassFIFO;
  FIFOF #(Bit #(32)) f_pipeline <- mkPipelineFIFO;

  // Producer rule (into f_bypass's enq side)
  rule rl_P;
    ... f_bypass.enq (x);
  endrule

  // Connect f_bypass's first/deq side
  // to f_pipeline's enq side
  mkConnection (to_FIFO_0 (f_bypass),
                to_FIFO_I (f_pipeline));

  // Consumer rule (from f_pipeline's first/deq side)
  rule rl_C;
    let y = f_pipeline.first;
    f_pipeline.deq;
  endrule
endmodule
```

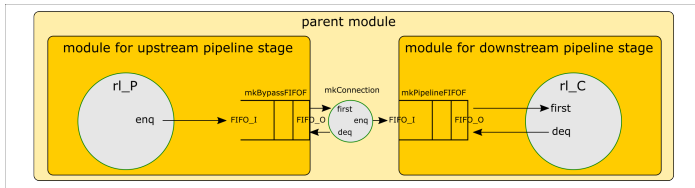


# BSV: Back-to-back composition of BypassFIFO and PipelineFIFO (2/2)

This has some pleasant properties:

- Despite their being two FIFOs, it takes only one tick to traverse them (because of the nature of BypassFIFOs).
- There are no ordering constraints across the composed FIFOs, *i.e.*, the FIFOs do not induce any ordering constraints between the producer rule `r1_P` and the consumer rule `r1_C` (they are “conflict free”). They can fire in the same clock and can go in either logical order.
- There are no combinational paths through the pair of FIFOs! One can verify this by studying the Verilog. The *bsc* compiler also helpfully reports the absence of combinational paths.

This makes it attractive for connecting stages in a pipeline, enabling a *modular* separation of stages, where each stage can be independently verified and compiled to Verilog. We use this technique everywhere in Fife.



# BSV: final comments on CRegs

- CRegs allow us to save a cycle by allowing two rules to run concurrently in the same clock where previously they had to run in separate clocks.

Effectively, CRegs allow us safely to *fuse* the actions in two different rules into a single composite action (whenever the rule conditions allow)

- It plays a central rule in fine-tuning **BSV** designs for optimal performance, without changing the functional semantics (rule-at-a-time).

In RISC-V designs, CRegs enable:

- Isolation of stages (no combinational paths) while preserving pipeline speed (as discussed in Slide 25).
- Faster PC redirection after a misprediction
- Faster resolution of register dependencies in the scoreboard
- Faster reorder buffers in Out-Of-Order processors
- ... and more ...

*Caveat: Before CRegs were available in **BSV**, designs used a facility called “RWires” for the same fusion optimizations. RWires are still available in BSV (see library documentation), but we recommend using CRegs in new designs. RWires muddy the otherwise clear separation of functional semantics (logical) and clocked semantics (implementation).*

End