Learn RISC-V CPU Implementation and **BSV**

(**BSV**: a High-Level Hardware Design Language)

Rishiyur S. Nikhil

L13: **BSV**: Rules and their semantics

bluespec

Reminders

Please git clone: `https://github.com/rsnikhil/Learn_Bluespec_and_RISCV_Design`
(git pull for latest version). Repsitory structure:

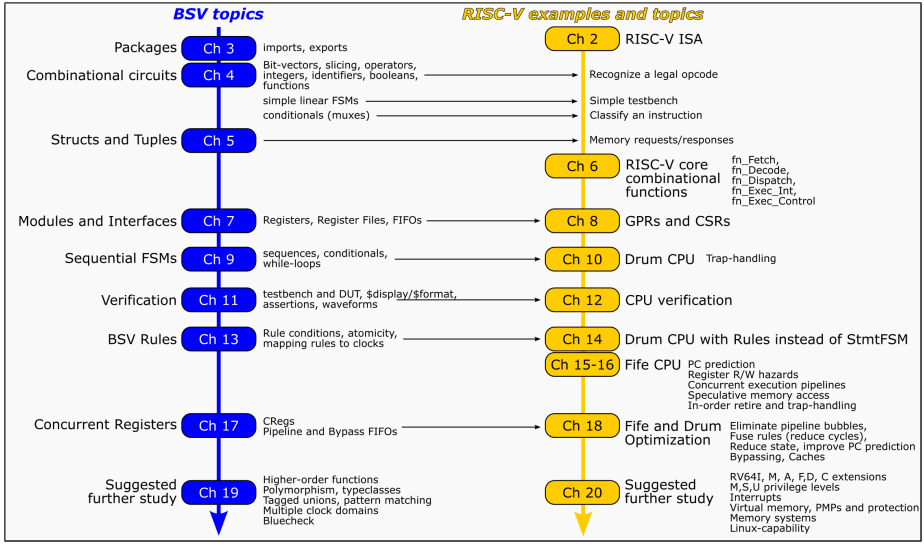
```
./Book_BLang_RISCV.pdf
  Slides/
      Slides_01_Intro.pdf
      Slides_02_ISA.pdf
      ...
  Exercises/
      Ex-03-A-Hello-World/
      Ex-03-B-Top-and-DUT/
      ...
  Code/
      src_Top/
      src_Drum/
      src_Fife/
      src_Common/
      ...
  Doc/Installing_bsc_Verilator_etc.{adoc,html}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

- 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



Flow of information between stages in Drum and Fife

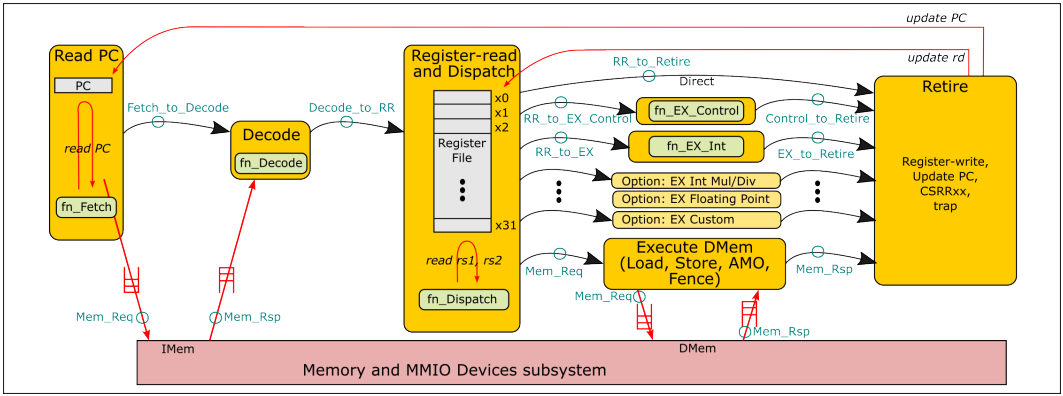


Table of Contents

Rules: Introduction

Rules: the fundamental behavioral construct in **BSV**

"*Rules*" are the fundamental constructs in **BSV** to specify dynamic behavior.

A `rule-endrule` construct appears in the body of a **BSV** modules.

It represents a combinational circuit that typically invokes methods in interfaces of other modules.

An interface method, in turn, is simply a "mini-rule" that is semantically a part of a rule or interface method from which it is invoked.

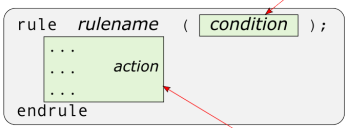
Thus, a rule consists of a `rule-endrule` construct plus every method it invokes, plus every method that those methods invoke, and so on. Thus, a rule can observe and update state in several modules.

The *functionality* of rules (semantics, "what does it do?") can be understood in two incremental steps:

- Semantics of a rule in isolation
- Semantics of the collection of rules in a **BSV** program

The *performance* of rules (how long does a computation take?) can be understood by understanding how rules are mapped to clocks.

Rules: Syntax, and data types for the two components



Both the explicit rule condition and the rule body may invoke other module methods, which may have implicit conditions. The explicit and implicit conditions together contribute to the overall rule firing condition (so-called `CAN_FIRE` *signal).*

A rule body is an implicit `action-endaction` block; it can contain several sub-actions and identifier bindings.

Rules: Implicit and overall condition

- Both the rule condition and the body may invoke interface methods of other modules.
- Each method has an "implicit condition" (type `Bool`; READY signal in hardware) indicating whether the method is currently enabled or not.

  E.g., for a standard FIFO f , the `f.first` and `f.deq` methods have implicit conditions that are true only when the FIFO is non-empty.
- A value-method like `f.first`, being pure (not `Action` or `ActionValue`), may be invoked both in rule conditions and in rule bodies.
- An `Action` or `ActionValue` method like `f.deq` can never be invoked in a rule condition, only in a rule body.

The overall rule condition, also known as its "CAN_FIRE" condition, is a conjunction (AND) of the rule's explicit condition and implicit conditions of any invoked methods, whether those methods are in the rule condition or in the rule body.

For example, if a rule invokes `f.first` or `f.deq`, the implicit condition of those methods become part of the CAN_FIRE condition of the rule.

Semantics of a rule in isolation

Semantics of a rule in isolation

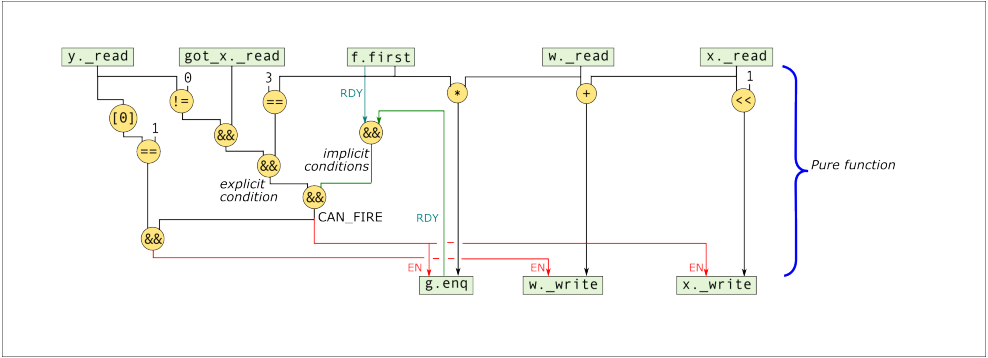
Each rule can be viewed as a pure function (therefore, a combinational circuit) whose inputs come from various methods and which produces outputs for various `Action` and `ActionValue` methods.

Each `Action` or `ActionValue` method has an implicit boolean ENABLE argument (separate from its normal arguments and result). An `Action` or `ActionValue` method *performs* its action only when its ENABLE argument is asserted.

Semantics of a rule in isolation

Example: the diagram below illustrates the function represented by this rule.

```
rule rl_compute ((y != 0) && got_x && (f.first == 3));
 if (y [0] == 1) w <= w + x;
 x <= x << 1;
 g.enq (w * f.first);
endrule
```



This is just a functional illustration; we will see shortly that this maps straightforwardly into hardware.

Recapping some properties of rules

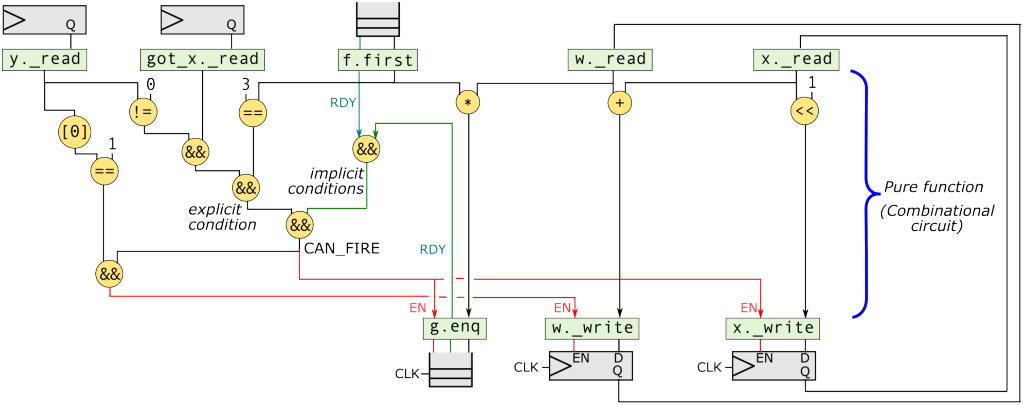
- All `Actions` in a rule are performed *simultaneously*, no matter what textual order they may appear in the rule body. We also say that the actions all occur *in parallel*.
- All `Actions` in a rule are performed *instantaneously*.
- Explicit rule conditions and method implicit conditions are combined to determine whether the rule executes at all.
- Some actions in a rule may be further restricted by if-then-else conditions in the rule.

Terminology: when a rule executes, we also say that the rule *fires*.

Hardware representation of a rule in isolation

The semantic view of a rule in Slide 12 maps straightforwardly into hardware.

(Green signals are READY signals; Red signals are ENABLE signals.)



Exercise break

Please see Book Appendix E, Section Ex-13-A-Simultaneous-Actions.

Note: a rule firing cannot perform the same action more than once

The following example illustrates some absurdities:

```
rule rl_foo (...);
   x <= 2;          x <= 3;
   f.enq (2);       f.enq (3);
   g.deq;           g.deq;
endrule
```

We cannot, in the same instant:

- write into a register more than once;
- enqueue into a FIFO more than once;
- dequeue from a FIFO more than once

The *bsc* compiler will flag such errors in a program with a message like:

"Cannot compose actions in parallel"

Semantics of a collection of rules

Semantics of a collection of rules

The semantics of a collection of **BSV** rules is: execute one-rule-at-a-time:

```
while True
    Choose any rule whose CAN_FIRE is true
        Perform the actions in that rule's body

        (this will likely modify state, affecting enabled CAN_FIRE signals for the next iteration)
```

By definition (because one-at-a-time):

- The state observed (reads) by a rule cannot be changing during rule execution
- The state updates (writes) by a rule cannot be observed by any other rule while it is changing.

We also say: rules are *atomic*

BSV *implementations* may (and do) execute rules concurrently and in parallel,
as long as it remains *consistent* with one-rule-at-a-time (atomic) semantics.

Analogies:

- RISC-V Instruction Semantics are one-instruction-at-a-time (instructions are atomic).
  This is the refence standard for "correct" behavior.
  RISC-V *implementations* may (and do) re-order instructions and execute multiple instructions at a time (pipelining, superscalarity).
- C/C++ semantics are one-statement-at-a-time (statements are atomic).
  This is the refence standard for "correct" behavior.
  C/C++ *implementations* may (and do) re-order and interleave instructions for different statements.

Semantics of rules: two sometimes surprising observations

NOTE:

The semantics of rules is non-deterministic—if CAN_FIRE is true for several rules, we can choose any one. This is common in formal specification languages for concurrent systems, allowing reasoning about correctness under *all* possible choices (different implementations may make different choices, statically or dynamically).

The *bsc* compiler makes particular choices, resulting in deterministic hardware.

NOTE:

Although **BSV** is a hardware design language (HDL), functional correctness of a **BSV** program only appeals to rule-at-a-time semantics, without any reference to clocks.

(Just as the functional correctness of a RISC-V implementation only appeals to one-instruction-at-a-time RISC-V ISA semantics, without any reference to clocks or cycles.)

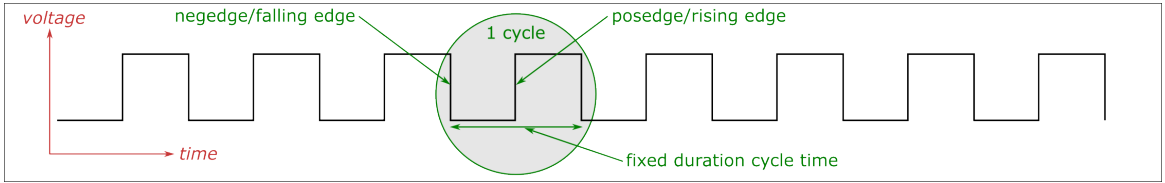
It is only for reasoning about performance ("how fast is the implementation?") that we need to appeal to clocks, cycles and implementation details.

Implementing rules in clocked digital hardware

Implementing rules in clocked digital hardware

Rule semantics provide only an abstract view of time: one rule executes *before* or *after* another, and we cannot ascribe any real-time measure (seconds, microseconds, nanoseconds) to rule executions.

Digital hardware is driven by a clock signal with a specific time-period:



The *bsc* compiler tries to execute as many rules as possible in each clock cycle (for maximum performance).

However, there are some constraints that limit this concurrency.

Constraints on concurrent execution of rules in a clock (1/2)

Suppose we want to implement each rule in hardware as suggested in the diagram in Slide 14.

Then, conceptually, at each posedge, all enabled rules (whose CAN_FIRE is true) will fire and perform their output actions.

But this would be wrong, for two reasons:

(1) Two rules may have *Action Conflicts* or *R*esource Conflicts if executed in the same clock.

Two rules may invoke the same `Action`/`ActionValue` method (write to the same register, or enqueue onto the same FIFO, dequeue the same FIFO, *etc.*). This is clearly not feasible at the same instant (same clock posedge).

Constraints on concurrent execution of rules in a clock (2/2)

(2) Two rules may have *Ordering Conflicts* if executed in the same clock, *i.e.,* the *ordering* can be inconsistent with rule semantics.

Consider these two rules, with x and y initially both having the value 10:

```
rule rl_r1 (...);
   x <= y + 1;
endrule
```

```
rule rl_r2 (...);
   y <= x + 2;
endrule
```

According to the one-rule-at-a-time semantics, rule `rl_r1` may precede `rl_r2`.
After `rl_r1`, `x` has the value 11. Then, after `rl_r2`, `y` has the value 13.

Or, according to the one-rule-at-a-time semantics, rule `rl_r2` may precede `rl_r1`.
After `rl_r2`, `y` has the value 12. Then, after `rl_r1`, `x` has the value 13.

In either case, register state-update by one rule is observed by the other rule.

Whereas, if we execute both rules in the same clock (at the same instant), neither rule observes the update by the other rule. `x` gets the value 11, and `y` gets the value 12. This is inconsistent with the rule-at-a-time semantics, and therefore an incorrect execution.

Rule arbitration and scheduling (1/2)

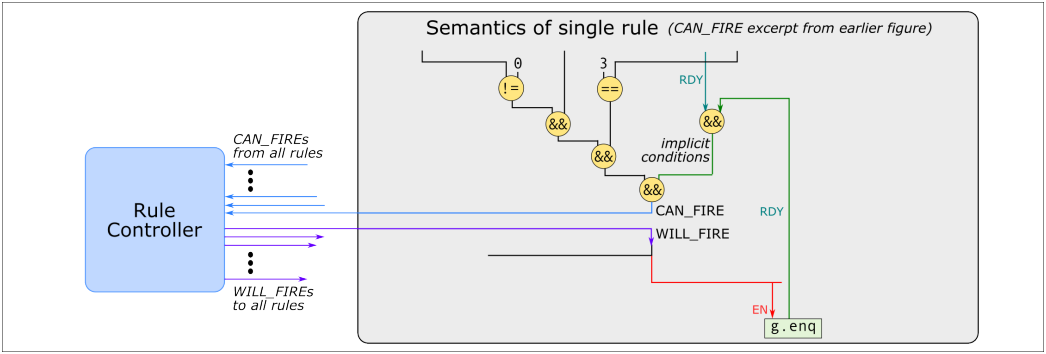
In summary, when two rules are enabled in the same clock and they have any of the conflicts discussed above, one of them needs to be *stalled* for that clock.

Analogy:

> Managing rule conflicts is similar to managing *write-write* and *read-write hazards* in processor pipelines, where instruction I_1 writes to a register and the next instruction I_2 reads from that register—we may have to stall I_2 until I_1 has performed its register-write.

Rule arbitration and scheduling (2/2)

Stalling a rule, when necessary, is performed by a *rule controller* (also sometimes called a *scheduler*).



This is a combinational circuit whose inputs are the `CAN_FIRE` signals from all rules in the design. For each `CAN_FIRE` input, it has a corresponding `WILL_FIRE` output signal, which connects to the ENABLE inputs of all the rule's actions.

The *bsc* compiler designs a custom rule controller for each design, based on an analysis of the actual rules in the design. It ensures that conflicting rules do not fire in the same clock.

Guiding *bsc*'s creation of the rule controller

When two rules with conflicts may both be enabled simultaneously (both CAN_FIREs true), the *bsc* compiler uses various heuristics to decide which one will be stalled by the generated rule controller.

bsc's choices are generally reasonable but, in rare cases, the programmer may wish to specify an explicit priority preference:

```
rule rl_r1 (...);
   ...
endrule

(* descending_urgency = "rl_r1, rl_r2" *)
rule rl_r2 (...);
   ...
endrule
```

This attribute forces the rule controller to stall rl_r2 whenever rl_r1 executes, whether or not they conflict:

```
(* preempts = "rl_r1, rl_r2" *)
```

If two rules' CAN_FIREs are mutually exclusive, they cannot conflict; this permits more efficient rule controller hardware. *bsc* is quite good at recognizing mutual exclusivity automatically, but the user can assist it explicitly:

```
(* mutually_exclusive = "rl_r1, rl_r2" *)
```

Rules: Final Comments

Rules: final comments

Rules are a unique feature of **BSV**.
We are unaware of any other HDL (Hardware Description Language) where behavior is specified using rules.

- Rules are the fundamental (and only) behavioral construct in **BSV**.
- Each rule is an *atomic transaction* modifying system state, which follows trivially from the rule-at-a-time semantics.

  Atomicity is one of the most powerful concepts in Computer Science for reasoning about correctness of concurrent systems.
- A rule comprises a syntactic `rule-endrule` construct; every interface method invoked by that rule; every method invoked by those methods; and so on. The overall rule condition (`CAN_FIRE`) encompasses the rule's explicit condition and the implicit conditions of all those methods.

  Thus, the atomicity of a rule is a *non-local* property—through its methods, it may observe and update state across many modules.
- The *bsc* compiler compiles the rules in a design into clocked digital hardware, attempting to maximize concurrency (number of rules that can execute in a clock) while remaining consistent with the rule-at-a-time semantics.

`StmtFSM` is just an EDSL[1] for Rules

[1] Embedded Domain-Specific Language

StmtFSM is just an EDSL for Rules (1/3)

Anything coded with StmtFSM can also be coded explicitly using rules. Example:

```
Stmt s = while (b)
            seq
               action1;
               if (c)
                  action2;
               else
                  action3;
               action4;
            endseq
```

StmtFSM is just an EDSL for Rules (2/3)

We can convert this into rules mechanically as follows:

- Each `Action` in the `Stmt` becomes the body of a rule.
- We add a register `rg_step` to control the sequencing. Each rule is enabled for a particu lar value of `rg_step`; when it executes, it updates `rg_step` to enable the next rule.

```
// rg_step == 7 means "idle"
// To start the FSM,
// environment sets rg_step <= 0
Reg #(Bit #(3)) rg_step <- mkReg (7);

rule rl_while (rg_step == 0);
   rg_step <= (b ? 1 : 100);
endrule

rule rl_A1 (rg_step == 1);
   action1;
   rg_step <= (c ? 2 : 3);  // if-then-else
endrule
```

```
rule rl_A2 (rg_step == 2);
   action2;
   rg_step <= 4;
endrule

rule rl_A3 (rg_step == 3);
   action3;
   rg_step <= 4;
endrule

rule rl_A4 (rg_step == 4);
   action4;
   rg_step <= 0;    // back to top of loop
endrule
```

StmtFSM is just an EDSL for Rules (3/3)

In summary, StmtFSM is just an "EDSL" (Embedded Domain-Specific Language) within **BSV** for *structured* rule flows (rule flows that can be characterized as a composition of sequencing, if-then-else, loops, and fork-join parallelism).

When we have *unstructured* rule flows, StmtFSM is not appropriate, and we use rules directly (*e.g.*, in Fife).

End